

Linux 环境编程 从应用到内核

高峰 李彬 著

Programming in Linux Environment
from Userspace to Kernel

- Linux领域第一本将应用编程与内核实现相结合的图书
- Linux环境编程的进阶指导，解析Linux接口的工作原理，帮助应用开发人员快速深入内核，掌握Linux系统运行机制



机械工业出版社
China Machine Press

Linux/Unix技术丛书

Linux环境编程：从应用到内核

高峰 李彬 著

ISBN: 978-7-111-53610-9

本书纸版由机械工业出版社于2016年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

前言

第0章 基础知识

0.1 一个Linux程序的诞生记

0.2 程序的构成

0.3 程序是如何“跑”的

0.4 背景概念介绍

0.4.1 系统调用

0.4.2 C库函数

0.4.3 线程安全

0.4.4 原子性

0.4.5 可重入函数

0.4.6 阻塞与非阻塞

0.4.7 同步与非同步

第1章 文件I/O

1.1 Linux中的文件

1.1.1 文件、文件描述符和文件表

1.1.2 内核文件表的实现

1.2 打开文件

1.2.1 open介绍

1.2.2 更多选项

1.2.3 open源码跟踪

1.2.4 如何选择文件描述符

1.2.5 文件描述符fd与文件管理结构file

1.3 creat简介

1.4 关闭文件

1.4.1 close介绍

1.4.2 close源码跟踪

1.4.3 自定义files_operations

1.4.4 遗忘close造成的问题

1.4.5 如何查找文件资源泄漏

1.5 文件偏移

1.5.1 lseek简介

1.5.2 小心lseek的返回值

1.5.3 lseek源码分析

1.6 读取文件

1.6.1 read源码跟踪

1.6.2 部分读取

1.7 写入文件

1.7.1 write源码跟踪

1.7.2 追加写的实现

1.8 文件的原子读写

1.9 文件描述符的复制

1.10 文件数据的同步

1.11 文件的元数据

1.11.1 获取文件的元数据

1.11.2 内核如何维护文件的元数据

1.11.3 权限位解析

1.12 文件截断

1.12.1 truncate与ftruncate的简单介绍

1.12.2 文件截断的内核实现

1.12.3 为什么需要文件截断

第2章 标准I/O库

2.1 stdin、stdout和stderr

2.2 I/O缓存引出的趣题

2.3 fopen和open标志位对比

2.4 fdopen与fileno

2.5 同时读写的痛苦

2.6 ferror的返回值

2.7 clearerr的用途

2.8 小心fgetc和getc

2.9 注意fread和fwrite的返回值

2.10 创建临时文件

第3章 进程环境

3.1 main是C程序的开始吗

3.2 “活雷锋”exit

3.3 atexit介绍

3.3.1 使用atexit

3.3.2 atexit的局限性

3.3.3 atexit的实现机制

3.4 小心使用环境变量

3.5 使用动态库

3.5.1 动态库与静态库

3.5.2 编译生成和使用动态库

3.5.3 程序的“平滑无缝”升级

3.6 避免内存问题

3.6.1 尴尬的realloc

3.6.2 如何防止内存越界

3.6.3 如何定位内存问题

3.7 “长跳转”longjmp

3.7.1 setjmp与longjmp的使用

3.7.2 “长跳转”的实现机制

3.7.3 “长跳转”的陷阱

第4章 进程控制：进程的一生

4.1 进程ID

4.2 进程的层次

4.2.1 进程组

4.2.2 会话

4.3 进程的创建之fork（）

4.3.1 fork之后父子进程的内存关系

4.3.2 fork之后父子进程与文件的关系

4.3.3 文件描述符复制的内核实现

4.4 进程的创建之vfork（）

4.5 daemon进程的创建

4.6 进程的终止

4.6.1 _exit函数

4.6.2 exit函数

4.6.3 return退出

4.7 等待子进程

4.7.1 僵尸进程

4.7.2 等待子进程之wait（）

4.7.3 等待子进程之waitpid（）

4.7.4 等待子进程之等待状态值

4.7.5 等待子进程之waitid（）

4.7.6 进程退出和等待的内核实现

4.8 exec家族

4.8.1 execve函数

4.8.2 exec家族

4.8.3 execve系统调用的内核实现

4.8.4 exec与信号

4.8.5 执行exec之后进程继承的属性

4.9 system函数

4.9.1 system函数接口

4.9.2 system函数与信号

4.10 总结

第5章 进程控制：状态、调度和优先级

5.1 进程的状态

5.1.1 进程状态概述

5.1.2 观察进程状态

5.2 进程调度概述

5.3 普通进程的优先级

5.4 完全公平调度的实现

5.4.1 时间片和虚拟运行时间

5.4.2 周期性调度任务

5.4.3 新进程的加入

5.4.4 睡眠进程醒来

5.4.5 唤醒抢占

5.5 普通进程的组调度

5.6 实时进程

5.6.1 实时调度策略和优先级

5.6.2 实时调度相关API

5.6.3 限制实时进程运行时间

5.7 CPU的亲合力

第6章 信号

6.1 信号的完整生命周期

6.2 信号的产生

6.2.1 硬件异常

6.2.2 终端相关的信号

6.2.3 软件事件相关的信号

6.3 信号的默认处理函数

6.4 信号的分类

6.5 传统信号的特点

6.5.1 信号的ONESHOT特性

6.5.2 信号执行时屏蔽自身的特性

6.5.3 信号中断系统调用的重启特性

6.6 信号的可靠性

6.6.1 信号的可靠性实验

6.6.2 信号可靠性差异的根源

6.7 信号的安装

6.8 信号的发送

6.8.1 kill、tkill和tkill

6.8.2 raise函数

6.8.3 sigqueue函数

6.9 信号与线程的关系

6.9.1 线程之间共享信号处理函数

6.9.2 线程有独立的阻塞信号掩码

6.9.3 私有挂起信号和共享挂起信号

6.9.4 致命信号下，进程组全体退出

6.10 等待信号

6.10.1 pause函数

6.10.2 sigsuspend函数

6.10.3 sigwait函数和sigwaitinfo函数

6.11 通过文件描述符来获取信号

6.12 信号递送的顺序

6.13 异步信号安全

6.14 总结

第7章 理解Linux线程（1）

7.1 线程与进程

7.2 进程ID和线程ID

7.3 pthread库接口介绍

7.4 线程的创建和标识

7.4.1 pthread_create函数

7.4.2 线程ID及进程地址空间布局

7.4.3 线程创建的默认属性

7.5 线程的退出

7.6 线程的连接与分离

7.6.1 线程的连接

7.6.2 为什么要连接退出的线程

7.6.3 线程的分离

7.7 互斥量

7.7.1 为什么需要互斥量

7.7.2 互斥量的接口

7.7.3 临界区的大小

7.7.4 互斥量的性能

7.7.5 互斥锁的公平性

7.7.6 互斥锁的类型

7.7.7 死锁和活锁

7.8 读写锁

7.8.1 读写锁的接口

7.8.2 读写锁的竞争策略

7.8.3 读写锁总结

7.9 性能杀手：伪共享

7.10 条件等待

7.10.1 条件变量的创建和销毁

7.10.2 条件变量的使用

第8章 理解Linux线程（2）

8.1 线程取消

8.1.1 函数取消接口

8.1.2 线程清理函数

8.2 线程局部存储

8.2.1 使用NPTL库函数实现线程局部存储

8.2.2 使用__thread关键字实现线程局部存储

8.3 线程与信号

8.3.1 设置线程的信号掩码

8.3.2 向线程发送信号

8.3.3 多线程程序对信号的处理

8.4 多线程与fork（）

第9章 进程间通信：管道

9.1 管道

9.1.1 管道概述

9.1.2 管道接口

9.1.3 关闭未使用的管道文件描述符

9.1.4 管道对应的内存区大小

9.1.5 shell管道的实现

9.1.6 与shell命令进行通信（popen）

9.2 命名管道FIFO

9.2.1 创建FIFO文件

9.2.2 打开FIFO文件

9.3 读写管道文件

9.4 使用管道通信的示例

第10章 进程间通信：System V IPC

10.1 System V IPC概述

10.1.1 标识符与IPC Key

10.1.2 IPC的公共数据结构

10.2 System V消息队列

10.2.1 创建或打开一个消息队列

10.2.2 发送消息

10.2.3 接收消息

10.2.4 控制消息队列

10.3 System V信号量

10.3.1 信号量概述

10.3.2 创建或打开信号量

10.3.3 操作信号量

10.3.4 信号量撤销值

10.3.5 控制信号量

10.4 System V共享内存

10.4.1 共享内存概述

10.4.2 创建或打开共享内存

10.4.3 使用共享内存

10.4.4 分离共享内存

10.4.5 控制共享内存

第11章 进程间通信：POSIX IPC

11.1 POSIX IPC概述

11.1.1 IPC对象的名字

11.1.2 创建或打开IPC对象

11.1.3 关闭和删除IPC对象

11.1.4 其他

11.2 POSIX消息队列

11.2.1 消息队列的创建、打开、关闭及删除

11.2.2 消息队列的属性

11.2.3 消息的发送和接收

11.2.4 消息的通知

11.2.5 I/O多路复用监控消息队列

11.3 POSIX信号量

11.3.1 创建、打开、关闭和删除有名信号量

11.3.2 信号量的使用

11.3.3 无名信号量的创建和销毁

11.3.4 信号量与futex

11.4 内存映射mmap

11.4.1 内存映射概述

11.4.2 内存映射的相关接口

11.4.3 共享文件映射

11.4.4 私有文件映射

11.4.5 共享匿名映射

11.4.6 私有匿名映射

11.5 POSIX共享内存

11.5.1 共享内存的创建、使用和删除

11.5.2 共享内存与tmpfs

第12章 网络通信：连接的建立

12.1 socket文件描述符

12.2 绑定IP地址

12.2.1 bind的使用

12.2.2 bind的源码分析

12.3 客户端连接过程

12.3.1 connect的使用

12.3.2 connect的源码分析

12.4 服务器端连接过程

12.4.1 listen的使用

12.4.2 listen的源码分析

12.4.3 accept的使用

12.4.4 accept的源码分析

12.5 TCP三次握手的实现分析

12.5.1 SYN包的发送

12.5.2 接收SYN包，发送SYN+ACK包

12.5.3 接收SYN+ACK数据包

12.5.4 接收ACK数据包，完成三次握手

第13章 网络通信：数据报文的发送

13.1 发送相关接口

13.2 数据包从用户空间到内核空间的流程

13.3 UDP数据包的发送流程

13.4 TCP数据包的发送流程

13.5 IP数据包的发送流程

13.5.1 ip_send_skb源码分析

13.5.2 ip_queue_xmit源码分析

13.6 底层模块数据包的发送流程

第14章 网络通信：数据报文的接收

14.1 系统调用接口

14.2 数据包从内核空间到用户空间的流程

14.3 UDP数据包的接收流程

14.4 TCP数据包的接收流程

14.5 TCP套接字的三个接收队列

14.6 从网卡到套接字

14.6.1 从硬中断到软中断

14.6.2 软中断处理

14.6.3 传递给协议栈流程

14.6.4 IP协议处理流程

14.6.5 大师的错误？原始套接字的接收

14.6.6 注册传输层协议

14.6.7 确定UDP套接字

14.6.8 确定TCP套接字

第15章 编写安全无错代码

15.1 不要用memcmp比较结构体

15.2 有符号数和无符号数的移位区别

15.3 数组和指针

15.4 再论数组首地址

15.5 “神奇”的整数类型转换

15.6 小心volatile的原子性误解

15.7 有趣的问题：“x==x”何时为假？

15.8 小心浮点陷阱

15.8.1 浮点数的精度限制

15.8.2 两个特殊的浮点值

15.9 Intel移位指令陷阱

前言

为什么要写这本书

我从事Linux环境的开发工作已有近十年的时间，但我一直认为工作时间并不等于经验，更不等于能力。如何才能把工作时间转换为自己的经验和能力呢？我认为无非是多阅读、多思考、多实践、多分享。这也是我在ChinaUnix上的博客座右铭，目前我的博客一共有247篇博文，记录的大都是Linux内核网络部分的源码分析，以及相关的应用编程。机械工业出版社华章公司的Lisa正是通过我的博客找到我的，而这也促成了本书的出版。

其实在Lisa之前，就有另外一位编辑与我聊过，但当时我没有下好决心，认为自己无论是在技术水平，还是时间安排上，都不足以完成一本技术图书的创作。等到与Lisa洽谈的时候，我感觉自己的技术已经有了一些沉淀，同时时间也相对比较充裕，因此决定开始撰写自己技术生涯的第一本书。

对于Linux环境的开发人员，《Unix环境高级编程》（后文均简称为APUE）无疑是最为经典的入门书籍。其作者Stevens是我从业以来最崇拜的技术专家。他的Advanced Programming in the Unix Environment、Unix Network Programming系列及TCP/IP Illustrated系列著作，字字珠玑，本本经典。在我从业的最初几年，这几本书每本都阅读了好几遍，而这也为我进行Linux用户空间的开发奠定了坚实的基础。在掌握了这些知识以后，如何继续提高自己的技能呢？经过一番思考，我选择了阅读Linux内核源码，并尝试将内核与应用融会贯通。在阅读了一定量的内核源码之后，我才真正理解了Linux专家的这句话“Read the fucking codes”。只有阅读了内核源码，才能真正理解Linux内核的原理和运行机制，而此时，我也发现了Stevens著作的一个局限——APUE和UNP毕竟是针对Unix环境而写的，Linux虽然大部分与Unix兼容，但是在很多行为上与Unix还是完全不同的。这就导致了书中的一些内容与Linux环境中的实际效果是相互矛盾的。

现在有机会来写一本技术图书，我就想在向Stevens致敬的同时，写一本类似于APUE风格的技术图书，同时还要在Linux环境下，对APUE进行突破。大言不惭地说，我期待这本书可以作为APUE的补充，还可以作为Linux开发人员的进阶读物。事实上，本书的写作布局正是以APUE的章节作为参考，针对Linux环境，不仅对用户空间的接口进行阐述，同时还引导读者分析该接口在内核的源码实现，使得读者不仅可以知道接口怎么用，同时还可以理解接口是怎么工作的。对于Linux的系统调用，做到知其然，知其所以然。

读者对象

根据本书的内容，我觉得适合以下几类读者：

- 在Linux应用层方面有一定开发经验的程序员。
- 对Linux内核有兴趣的程序员。
- 热爱Linux内核和开源项目的技术人员。

如何阅读本书

本书定位为APUE的补充或进阶读物，所以假设读者已具备了一定的编程基础，对Linux环境也有所了解，因此在涉及一些基本概念和知识时，只是蜻蜓点水，简单略过。因为笔者希望把更多的笔墨放在更为重要的部分，而不是各种相关图书均有讲解的基本概念上。所以如果你是初学者，建议还是先学习APUE、C语言编程，并且在具有一定的操作系统知识后再来阅读本书。

Linux环境编程涉及的领域太多，很难有某个人可以在Linux的各个领域均有比较深刻的认识，尤其是已有APUE这本经典图书在前，所以本书是由高峰、李彬两个人共同完成的。

高峰负责第0、1、2、3、4、12、13、14、15章，李彬负责第5~11章。两位不同的作者，在写作风格上很难保证一致，如果给各位读者带来了不便，在此给各位先道个歉。尽管是由两个人共同写作，并且负责的还是我们各自相对擅长的领域，可是在写作的过程中我们仍然感觉到很吃力，用了将近三年的时间才算完成本书。对比APUE，本书一方面在深度上还是有所不及，另一方面在广度上还是没有涵盖APUE涉及的所有领域，这也让我们对Stevens大师更加敬佩。

本书使用的Linux内核源代码版本为3.2.44，glibc的源码版本为2.17。

勘误和支持

由于作者的水平有限，主题又过于宏大，书中难免会出现一些错误或不准确的地方，如有不妥之处，恳请读者批评指正。如果你发现有什么问题，或者有什么疑问，都可以发邮件至我的邮箱 gfree.wind@gmail.com，期待您的指导！

致谢

首先要感谢伟大的Linux内核创始人Linus，他开创了一个影响世界的操作系统。

其次要感谢机械工业出版社华章公司的编辑杨绣国老师（Lisa），感谢你的魄力，敢于找新人来写

作，并敢于信任新人，让其完成这么大的一个项目。感谢你的耐心，正常的一年半的写作时间，被我们生生地延长到了将近三年的时间，感谢你在写作过程中对我们的鼓励和帮助。

然后要感谢我的搭档李彬，在我加入当前的创业公司后，只有很少的空闲时间和精力来投入写作。这时，是李彬在更紧张的时间内，承担了本书的一半内容。并且其写作态度极其认真，对质量精益求精。没有李彬的加入，本书很可能就半途而废了。再次感谢李彬，我的好搭档。

最后我要感谢我的亲人。感谢我的父母，没有你们的培养，绝没有我的今天；感谢我的妻子，没有你的支持，就没有我事业上的进步；感谢我的岳父岳母对我女儿的照顾，使我没有后顾之忧；最后要感谢的是我可爱的女儿高一涵小天使，你的诞生为我带来了无尽的欢乐和动力！

谨以此书，献给我最亲爱的家人，以及众多热爱Linux的朋友们。

高峰

中国北京

2016年3月

第0章 基础知识

基础知识是构建技术大厦不可或缺的稳定基石，因此，本书首先来介绍一下书中所涉及的一些基础知识。这里以第0章命名，表明我们要注重基础，从0开始，同时也是向伟大的C语言致敬。

基础知识看似简单，但是想要真正理解它们，是需要花一番功夫的。除了需要积累经验以外，更需要对它们进行不断的思考和理解，这样，才能写出高可靠性的程序。这些基础知识很多都可以独立成文，限于篇幅，这里只能是简单的介绍，都是笔者根据自己的经验和理解进行的总结和概括，相信对读者会有所帮助。感兴趣的朋友可以自己查找更多的资料，以得到更准确、更细致的介绍。



注意 本书中的示例代码为了简洁明了，没有考虑代码的健壮性，例如不检查函数的返回值、使用全局变量等。

0.1 一个Linux程序的诞生记

一本编程书籍如果开篇不写一个“hello world”，就违背了“自古以来”的传统了。因此本节也将以hello world为例来说明一个Linux程序的诞生过程，示例代码如下：

```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

下面使用gcc生成可执行程序：gcc-g-Wall 0_1_hello_world.c-o hello_world。这样，一个Linux可执行程序就诞生了。

整个过程看似简单，其实涉及预处理、编译、汇编和链接等多个步骤。只不过gcc作为一个工具集自动完成了所有的步骤。下面就分别来看看其中所涉及各个步骤。

首先来了解一下什么是预处理。预处理用于处理预处理命令。对于上面的代码来说，唯一的预处理命令就是#include。它的作用是将头文件的内容包含到本文件中。注意，这里的“包含”指的是该头文件中的所有代码都会在#include处展开。可以通过“gcc-E 0_1_hello_world.c”在预处理后自动停止后面的操作，并把预处理的结果输出到标准输出。因此使用“gcc-E 0_1_hello_world.c>0_1_hello_world.i”，可得到预处理后的文件。

理解了预处理，在出现一些常见的错误时，才能明白其中的原因。比如，为什么不能在头文件中定义全局变量？这是因为定义全局变量的代码会存在于所有以#include包含该头文件的文件中，也就是说所有的这些文件，都会定义一个同样的全局变量，这样就不可避免地造成了冲突。

编译环节是指对源代码进行语法分析，并优化产生对应的汇编代码的过程。同样，可以使用gcc得到汇编代码，而非最终的二进制文件，即“gcc-S 0_1_hello_world.c-o 0_1_hello_world.s”。gcc的-S选项会让gcc在编译完成后停止后面的工作，这样只会产生对应的汇编文件。

汇编的过程比较简单，就是将源代码翻译成可执行的指令，并生成目标文件。对应的gcc命令为“gcc-c 0_1_hello_world.c-o 0_1_hello_world.o”。

链接是生成最终可执行程序的最后一步，也是比较复杂的一步。它的工作就是将各个目标文件——包括库文件（库文件也是一种目标文件）链接成一个可执行程序。在这个过程中，涉及的概念比较多，如地址和空间的分配、符号解析、重定位等。在Linux环节下，该工作是由GNU的链接器ld完成的。

实际上我们可以使用-v选项来查看完整和详细的gcc编译过程，命令如下。

```
gcc -g -Wall -v 0_1_hello_word.c -o hello_world.
```

由于输出过多，此处就不粘贴结果了。感兴趣的朋友可以自行执行命令，查看输出。通过-v选项，可以看到gcc在背后做了哪些具体的工作。

0.2 程序的构成

Linux下二进制可执行程序的一般格式为ELF格式。以0.1节的hello world为例，使用readelf查看其ELF格式，内容如下：

```
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8048320
Start of program headers: 52 (bytes into file)
Start of section headers: 5148 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 9
Size of section headers: 40 (bytes)
Number of section headers: 36
Section header string table index: 33
Section Headers:
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048154 000154 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048168 000168 000020 00 A 0 0 4
[ 3] .note.gnu.build-id NOTE 08048188 000188 000024 00 A 0 0 4
[ 4] .gnu.hash GNU_HASH 080481ac 0001ac 000020 04 A 5 0 4
[ 5] .dynsym DYNsym 080481cc 0001cc 000050 10 A 6 1 4
[ 6] .dynstr STRTAB 0804821c 00021c 00004a 00 A 0 0 1
[ 7] .gnu.version VERSYM 08048266 000266 00000a 02 A 5 0 2
[ 8] .gnu.version_r VERNEED 08048270 000270 000020 00 A 6 1 4
[ 9] .rel.dyn REL 08048290 000290 000008 08 A 5 0 4
[10] .rel.plt REL 08048298 000298 000018 08 A 5 12 4
[11] .init PROGBITS 080482b0 0002b0 000024 00 AX 0 0 4
[12] .plt PROGBITS 080482e0 0002e0 000040 04 AX 0 0 16
[13] .text PROGBITS 08048320 000320 000188 00 AX 0 0 16
[14] .fini PROGBITS 080484a8 0004a8 000015 00 AX 0 0 4
[15] .rodata PROGBITS 080484c0 0004c0 000015 00 A 0 0 4
[16] .eh_frame_hdr PROGBITS 080484d8 0004d8 000034 00 A 0 0 4
[17] .eh_frame PROGBITS 0804850c 00050c 0000c4 00 A 0 0 4
[18] .init_array INIT_ARRAY 08049f08 000f08 000004 00 WA 0 0 4
[19] .fini_array FINI_ARRAY 08049f0c 000f0c 000004 00 WA 0 0 4
[20] .jcr PROGBITS 08049f10 000f10 000004 00 WA 0 0 4
[21] .dynamic DYNAMIC 08049f14 000f14 0000e8 08 WA 6 0 4
[22] .got PROGBITS 08049ffc 000ffc 000004 04 WA 0 0 4
[23] .got.plt PROGBITS 0804a000 001000 000018 04 WA 0 0 4
[24] .data PROGBITS 0804a018 001018 000008 00 WA 0 0 4
[25] .bss NOBITS 0804a020 001020 000004 00 WA 0 0 4
[26] .comment PROGBITS 00000000 001020 00006b 01 MS 0 0 1
[27] .debug_aranges PROGBITS 00000000 00108b 000020 00 0 0 1
[28] .debug_info PROGBITS 00000000 0010ab 000094 00 0 0 1
[29] .debug_abbrev PROGBITS 00000000 00113f 000044 00 0 0 1
[30] .debug_line PROGBITS 00000000 001183 000043 00 0 0 1
[31] .debug_str PROGBITS 00000000 0011c6 0000cb 01 MS 0 0 1
[32] .debug_loc PROGBITS 00000000 001291 000038 00 0 0 1
[33] .shstrtab STRTAB 00000000 0012c9 000151 00 0 0 1
[34] .symtab SYMTAB 00000000 0019bc 000490 10 35 51 4
[35] .strtab STRTAB 00000000 001e4c 00025a 00 0 0 1
```

由于输出过多，后面的结果并没有完全展示出来。ELF文件的主要内容就是由各个section及symbol表组成的。在上面的section列表中，大家最熟悉的应该是text段、data段和bss段。text段为代码段，用于保存可执行指令。data段为数据段，用于保存有非0初始值的全局变量和静态变量。bss段用于保存没有初始值或初值为0的全局变量和静态变量，当程序加载时，bss段中的变量会被初始化为0。这个段并不占用物理空间——因为完全没有必要，这些变量的值固定初始化为0，因此何必占用宝贵的物理空间？

其他段没有这三个段有名，下面来介绍一下其中一些比较常见的段：

·**debug**段：顾名思义，用于保存调试信息。

·**dynamic**段：用于保存动态链接信息。

·**fini**段：用于保存进程退出时的执行程序。当进程结束时，系统会自动执行这部分代码。

·**init**段：用于保存进程启动时的执行程序。当进程启动时，系统会自动执行这部分代码。

·**rodata**段：用于保存只读数据，如**const**修饰的全局变量、字符串常量。

·**symtab**段：用于保存符号表。

其中，对于与调试相关的段，如果不使用**-g**选项，则不会生成，但是与符号相关的段仍然会存在，这时可以使用**strip**去掉符号信息，感兴趣的朋友可以自己参考**strip**的说明进行实验。一般在嵌入式的产品中，为了减少程序占用的空间，都会使用**strip**去掉非必要的段。

0.3 程序是如何“跑”的

在日常工作中，我们经常会说“程序“跑”起来了”，那么它到底是怎么“跑”的呢？在Linux环境下，可以使用strace跟踪系统调用，从而帮助自己研究系统程序加载、运行和退出的过程。此处仍然以hello_world为例。

```
strace ./hello world
execve("./hello world", [ "./hello world" ], [ /* 59 vars */ ]) = 0
brk(0) = 0x872a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7778000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80063, ...}) = 0
mmap2(NULL, 80063, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7764000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\000\226\1\0004\0\0\0"... , 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1730024, ...}) = 0
mmap2(NULL, 1743580, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75ba000
mprotect(0xb75ba000, 4096, PROT_NONE) = 0
mmap2(0xb775e000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a3) = 0xb775e000
mmap2(0xb7761000, 10972, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7761000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb75b9000
set_thread_area({entry_number:-1-> 6, base_addr:0xb75b9900, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb775e000, 8192, PROT_READ) = 0
mprotect(0x8049000, 4096, PROT_READ) = 0
mprotect(0xb779b000, 4096, PROT_READ) = 0
munmap(0xb7764000, 80063) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7777000
write(1, "Hello world!\n", 13Hello world!
) = 13
exit_group(0) = ?
```

下面就针对strace输出说明其含义。在Linux环境中，执行一个命令时，首先是由shell调用fork，然后在子进程中来真正执行这个命令（这一过程在strace输出中无法体现）。strace是hello_world开始执行后的输出。首先是调用execve来加载hello_world，然后ld会分别检查ld.so.nohwcap和ld.so.preload。其中，如果ld.so.nohwcap存在，则ld会加载其中未优化版本的库。如果ld.so.preload存在，则ld会加载其中的库——在一些项目中，我们需要拦截或替换系统调用或C库，此时就会利用这个机制，使用LD_PRELOAD来实现。之后利用mmap将ld.so.cache映射到内存中，ld.so.cache中保存了库的路径，这样就完成了所有的准备工作。接着ld加载c库——libc.so.6，利用mmap及mprotect设置程序的各个内存区域，到这里，程序运行的环境已经完成。后面的write会向文件描述符1（即标准输出）输出"Hello world! \n"，返回值为13，它表示write成功的字符个数。最后调用exit_group退出程序，此时参数为0，表示程序退出的状态——此例中hello-world程序返回0。

0.4 背景概念介绍

0.4.1 系统调用

系统调用是操作系统提供的服务，是应用程序与内核通信的接口。在x86平台上，有多种陷入内核的途径，最早是通过`int 0x80`指令来实现的，后来Intel增加了一个新的指令`sysenter`来代替`int 0x80`——其他CPU厂商也增加了类似的指令。新指令`sysenter`的性能消耗大约是`int 0x80`的一半左右。即使是这样，相对于普通的函数调用来说，系统调用的性能消耗也是巨大的。所以在追求极致性能的程序中，都在尽力避免系统调用，譬如C库的`gettimeofday`就避免了系统调用。

用户空间的程序默认是通过栈来传递参数的。对于系统调用来说，内核态和用户态使用的是不同的栈，这使得系统调用的参数只能通过寄存器的方式进行传递。

有心的朋友可能会想到一个问题：在写代码的时候，程序员根本不用关心参数是如何传递的，编译器已经默默地为我们做了一切——压栈、出栈、保存返回地址等操作，但是编译器如何知道调用的函数是普通函数，还是系统调用呢？如果是后者，编译器就不能简单地使用栈来传递参数了。

为了解决这个问题，我们就要看0.4.2节介绍的C库函数了。

0.4.2 C库函数

0.4.1节提到C库函数为编译器解决了系统调用的问题。Linux环境下，使用的C库一般都是glibc，它封装了几乎所有的系统调用，代码中使用的“系统调用”，实际上就是调用C库中的函数。C库函数同样位于用户态，所以编译器可以统一处理所有的函数调用，而不用区分该函数到底是不是系统调用。

下面以具体的系统调用open来看看glibc库是如何封装系统调用的。在glibc的代码中，用了大量的编译器特性以及编程的技巧，可读性不高。open在glibc中对应的实现函数实际上是__open_nocancel。至于如何定位到它，感兴趣的朋友可以用__open_nocancel或open作为关键字，在glibc的源码中搜索，找出它们之间的关系。

```
int
__open_nocancel (const char *file, int oflag, ...)
{
    int mode = 0;
    if (oflag & O_CREAT)
    {
        va_list arg;
        va_start (arg, oflag);
        mode = va_arg (arg, int);
        va_end (arg);
    }
    return INLINE_SYSCALL (openat, 4, AT_FDCWD, file, oflag, mode);
}
```

其中INLINE_SYSCALL是我们关心的内容，这个宏完成了对真正系统调用的封装：INLINE_SYSCALL->INTERNAL_SYSCALL。实现INTERNAL_SYSCALL的一个实例为。

```
# define INTERNAL_SYSCALL(name, err, nr, args...)          \
({                                                           \
    register unsigned int resultvar;                        \
    EXTRAVAR_##nr                                           \
    asm volatile (                                          \
        LOADARGS_##nr                                       \
        "movl %1, %%eax\n\t"                                \
        "int $0x80\n\t"                                       \
        RESTOREARGS_##nr                                    \
        : "=a" (resultvar)                                  \
        : "i" (__NR_##name) ASMFMT_##nr(args) : "memory", "cc"); \
    (int) resultvar; })
```

其中，关键的代码是用嵌入式汇编写的，在此只做简单说明。“move%1, %%eax”表示将第一个参数（即__NR_##name）赋给寄存器eax。__NR_##name为对应的系统调用号，对于本例中的open来说，其为__NR_openat。系统调用号在文件/usr/include/asm/unitstd_32（64）.h中定义，代码如下：

```
[fgao@fgao understanding_apue]#cat /usr/include/asm/unistd_32.h | grep openat
#define __NR_openat 295
```

也就是说，在Linux平台下，系统调用的约定是使用寄存器eax来传递系统调用号的。至于参数的传递，在glibc中也有详细的说明，参见文件sysdeps/unix/sysv/linux/i386/sysdep.h。

0.4.3 线程安全

线程安全，顾名思义是指代码可以在多线程环境下“安全”地执行。何谓安全？即符合正确的逻辑结果，是程序员期望的正常执行结果。为了实现线程安全，该代码要么只能使用局部变量或资源，要么就是利用锁等同步机制，来实现全局变量或资源的串行访问。

下面是一个经典的多线程不安全代码：

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
static int counter = 0;
#define LOOPS 10000000
static void * thread(void * unused)
{
    int i;
    for (i = 0; i < LOOPS; ++i) {
        ++counter;
    }
    return NULL;
}
int main(void)
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread, NULL);
    pthread_create(&t2, NULL, thread, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter is %d by threads\n", counter);
    return 0;
}
```



注意 之所以这里的LOOPS选了一个比较大的数“10000000”，是为了保证第一个线程不要在第二个线程开始执行前就退出了。大家可以根据自己的运行环境来修改这个数值。

以上代码创建了两个线程，用来实现对同一个全局变量进行自加运算，循环次数为一千万次。下面来看一下运行结果：

```
[fgao@fgao chapter0]# ./threads_counter
Counter is 10843915 by threads
```

为什么最后的结果不是期望的20000000呢？下面反汇编将来揭开这个秘密——反汇编是理解程序行为的不二利器，因为它更贴近机器语言，也就是说，反汇编更贴近CPU运行的真相。

下面对线程函数thread进行反汇编，代码如下：

```
080484a4 <thread>:
80484a4: 55                push    %ebp
80484a5: 89 e5             mov     %esp,%ebp
80484a7: 83 ec 10         sub     $0x10,%esp
80484aa: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%ebp)
80484b1: eb 11            jmp     80484c4 <thread+0x20>80484b3:
```

```

mov

0x8049894,%eax

80484b8:

83 c0 01

add

$0x1,%eax

80484bb:

a3 94 98 04 08

mov

%eax,0x8049894

80484c0:      83 45 fc 01      addl    $0x1,-0x4(%ebp)
80484c4:      81 7d fc 7f 96 98 00    cmpl    $0x98967f,-0x4(%ebp)
80484cb:      7e e6            jle     80484b3 <thread+0xf>
80484cd:      b8 00 00 00 00    mov     $0x0,%eax
80484d2:      c9              leave
80484d3:      c3              ret

```

其中加粗部分对应的是++counter的汇编代码，其逻辑如下：

- 1) 将counter的值赋给寄存器EAX;
- 2) 对寄存器EAX的值加1;
- 3) 将EAX的值赋给counter。

假设目前counter的值为0，那么当两个线程同时执行++counter时，会有如下情况（每个线程会有独立的上下文执行环境，所以可视为每个线程都有一个“独立”的EAX）：

thread1	thread2
eax = counter => eax = 0	eax = counter => 0 eax = 0
eax = eax+1 => eax = 1	
counter = eax => counter = 1	

```
eax = eax + 1 => eax = 1  
counter = eax => counter = 1
```

上面两个线程都对**counter**执行了自增动作，但是最终的结果是“1”而不是“2”。这只是众多错误时序情况中的一种。之所以会产生这样的错误，就是因为++**counter**的执行指令并不是原子的，多个线程对**counter**的并发访问造成了最后的错误结果。利用锁就可以保证**counter**自增指令的串行化，如下所示：

```
thread1                                thread2  
lock  
eax = counter => eax = 0  
eax = eax + 1 => eax = 1  
counter = eax => counter = 1  
unlock  
  
lock  
eax = counter => eax = 1  
eax = eax + 1 => eax = 2  
counter = eax => counter = 2  
unlock
```

通过加锁，可以视**counter**的自增指令为“原子指令”，最后的结果终于是期望的答案了。

0.4.4 原子性

以前原子被认为是物理组成的最小单元，所以在计算机领域，就借其不可分割的这层含义作为隐喻。对于计算机科学来说，如果变量是原子的，那么对这个变量的任何访问和更改都是原子的。如果操作是原子的，那么这个操作将是不可分割的，要么成功，要么失败，不会有任何的中间状态。

列举一个原子操作的例子，用户A向用户B转账1000元。简单来说，这里最起码有两个步骤：

- 1) 用户A的账号减少1000元；
- 2) 用户B的账号增加1000元。

如果在上述步骤1结束的时候，转账发生了故障，比如电力中断，是否会造成用户A的账号减少了1000元，而用户B的账号没有变化呢？这种情况对于原子操作是不会发生的。当电力中断导致转账操作进行到一半就失败时，用户A的账号肯定不会减少1000元。因为这个操作的原子性，保证了用户A减少1000元和用户B增加1000元，必须同时成立，而不会存在一个中间结果。至于这个操作是如何做到原子性的，可以参看数据库的事务是如何实现的——原子性是事务的一个特性之一。

0.4.5 可重入函数

从字面上理解，可重入就是可重复进入。在编程领域，它不仅仅意味着可以重复进入，还要求在进入后能成功执行。这里的重复进入，是指当前进程已经处于该函数中，这时程序会允许当前进程的某个执行流程再次进入该函数，而不会引发问题。这里的执行流程不仅仅包括多线程，还包括信号处理、`longjump`等执行流程。所以，可重入函数一定是线程安全的，而线程安全函数则不一定是可重入函数。

从以上定义来看，很难说出哪些函数是可重入函数，但是可以很明显看出哪些函数是不可以重入的函数。当函数使用锁的时候，尤其是互斥锁的时候，该函数是不可重入的，否则会造成死锁。若函数使用了静态变量，并且其工作依赖于这个静态变量时，该函数也是不可重入的，否则会造成该函数工作不正常。

下面来看一个死锁的例子代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static const char * const caller[2] = {"mutex_thread", "signal handler"};
static pthread_t mutex_tid;
static pthread_t sleep_tid;
static volatile int signal_handler_exit = 0;
static void hold_mutex(int c)
{
    printf("enter hold_mutex [caller %s]\n", caller[c]);
    pthread_mutex_lock(&mutex);
    /* 这里的循环是为了保证锁不会在信号处理函数退出前被释放掉 */

    /*
    while (!signal_handler_exit && c != 1) {
        sleep(5);
    }
    pthread_mutex_unlock(&mutex);
    printf("leave hold_mutex [caller %s]\n", caller[c]);
*/
}
static void *mutex_thread(void *arg)
{
    hold_mutex(0);
}
static void *sleep_thread(void *arg)
{
    sleep(10);
}
static void signal_handler(int signum)
{
    hold_mutex(1);
    signal_handler_exit = 1;
}
int main()
{
    signal(SIGUSR1, signal_handler);
    pthread_create(&mutex_tid, NULL, mutex_thread, NULL);
    pthread_create(&sleep_tid, NULL, sleep_thread, NULL);
    pthread_kill(sleep_tid, SIGUSR1);
    pthread_join(mutex_tid, NULL);
    pthread_join(sleep_tid, NULL);
    return 0;
}
```

先看看运行结果：

```
[fgao@fgao chapter0]#gcc -g 0_8_signal_mutex.c -o signal_mutex -lpthread
[fgao@fgao chapter0]#./signal_mutex
enter hold_mutex [caller signal handler]
enter hold_mutex [caller mutex_thread]
```

为什么会死锁呢？就是因为函数`hold_mutex`是不可重入的函数——其中使用了`pthread_mutex`互斥量。当`mutex_thread`获得`mutex`时，`sleep_thread`就收到了信号，再次调用就进入了`hold_mutex`。结果始终无法拿到`mutex`，信号处理函数无法返回，正常的程序流程也无法继续，这就造成了死锁。

0.4.6 阻塞与非阻塞

这里的阻塞与非阻塞，都是指I/O操作。在Linux环境下，所有的I/O系统调用默认都是阻塞的。那么何谓阻塞呢？阻塞的系统调用是指，当进行系统调用时，除非出错（被信号打断也视为出错），进程将会一直陷入内核态直到调用完成。非阻塞的系统调用是指无论I/O操作成功与否，调用都会立刻返回。

0.4.7 同步与非同步

这里的同步与非同步，也是指I/O操作。当把阻塞、非阻塞、同步和非同步放在一起时，不免会让人眼花缭乱。同步是否就是阻塞，非同步是否就是非阻塞呢？实际上在I/O操作中，它们是不同的概念。同步既可以是阻塞的，也可以是非阻塞的，而常用的Linux的I/O调用实际上都是同步的。这里的同步和非同步，是指I/O数据的复制工作是否同步执行。

以系统调用read为例。阻塞的read会一直陷入内核态直到read返回；而非阻塞的read在数据未准备好的情况下，会直接返回错误，而当有数据时，非阻塞的read同样会一直陷入内核态，直到read完成。这个read就是同步的操作，即I/O的完成是在当前执行流程下同步完成的。

如果是非同步即异步，则I/O操作不是随系统调用同步完成的。调用返回后，I/O操作并没有完成，而是由操作系统或者某个线程负责真正的I/O操作，等完成后通知原来的线程。

第1章 文件I/O

文件I/O是操作系统不可或缺的部分，也是实现数据持久化的手段。对于Linux来说，其“一切皆是文件”的思想，更是突出了文件在Linux内核中的重要地位。本章主要讲述Linux文件I/O部分的系统调用。



注意 为了分析系统调用的实现，从本章开始会涉及Linux内核源码。但是本书并不是一本介绍内核源码的书籍，所以书中对内核源码的分析不会面面俱到。分析内核源码的目的是为了更好地理解系统调用，是为应用而服务的。因此，本书对内核源码的追踪和分析，只是浅尝辄止。

1.1 Linux中的文件

1.1.1 文件、文件描述符和文件表

Linux内核将一切视为文件，那么Linux的文件是什么呢？其既可以是事实上的真正的物理文件，也可以是设备、管道，甚至还可以是一块内存。狭义的文件是指文件系统中的物理文件，而广义的文件则可以是Linux管理的所有对象。这些广义的文件利用VFS机制，以文件系统的形式挂载在Linux内核中，对外提供一致的文件操作接口。

从数值上看，文件描述符是一个非负整数，其本质就是一个句柄，所以也可以认为文件描述符就是一个文件句柄。那么何为句柄呢？一切对于用户透明的返回值，即可视为句柄。用户空间利用文件描述符与内核进行交互；而内核拿到文件描述符后，可以通过它得到用于管理文件的真正的数据结构。

使用文件描述符即句柄，有两个好处：一是增加了安全性，句柄类型对用户完全透明，用户无法通过任何hacking的方式，更改句柄对应的内部结果，比如Linux内核的文件描述符，只有内核才能通过该值得到对应的文件结构；二是增加了可扩展性，用户的代码只依赖于句柄的值，这样实际结构的类型就可以随时发生变化，与句柄的映射关系也可以随时改变，这些变化都不会影响任何现有的用户代码。

Linux的每个进程都会维护一个文件表，以便维护该进程打开文件的信息，包括打开的文件个数、每个打开文件的偏移量等信息。

1.1.2 内核文件表的实现

内核中进程对应的结构是`task_struct`，进程的文件表保存在`task_struct->files`中。其结构代码如下所示。

```
struct files_struct {  
    /* count为文件表
```

`files_struct`的引用计数

```
*/  
    atomic_t count;  
    /* 文件描述符表
```

```
*/  
    /*  
        为什么有两个
```

`fdtable`呢？这是内核的一种优化策略。

`fdt`为指针，而

`fdtab`为普通变量。一般情况下，

`fdt`是指向

`fdtab`的，当需要它的时候，才会真正动态申请内存。因为默认大小的文件表足以应付大多数

情况，因此这样就可以避免频繁的内存申请。

这也是内核的常用技巧之一。在创建时，使用普通的变量或者数组，然后让指针指向它，作为默认情况使

用。只有当进程使用量超过默认值时，才会动态申请内存。

```
*/  
    struct fdtable __rcu *fdt;  
    struct fdtable fdtab;  
    /*  
    * written part on a separate cache line in SMP  
    */  
    /* 使用
```

____cacheline_aligned_in_smp可以保证

file_lock是以

cache
line 对齐的，避免了

```
false sharing */
spinlock_t file_lock ____cacheline_aligned_in_smp;
/* 用于查找下一个空闲的
```

```
fd */
int next_fd;
/* 保存执行
```

exec需要关闭的文件描述符的位图

```
*/
struct embedded_fd_set close_on_exec_init;
/* 保存打开的文件描述符的位图
```

```
*/
struct embedded_fd_set open_fds_init;
/* fd_array为一个固定大小的
```

file结构数组。

struct file是内核用于文

件管理的结构。这里使用默认大小的数组，就是为了可以涵盖大多数情况，避免动

态分配

```
*/
struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
```

下面看看files_struct是如何使用默认的fdtab和fd_array的，init是Linux的第一个进程，它的文件表是一个全局变量，代码如下：

```
struct files_struct init_files = {
    .count      = ATOMIC_INIT(1),
    .fdt        = &init_files.fdtab,
    .fdtab      = {
        .max_fds = NR_OPEN_DEFAULT,
        .fd      = &init_files.fd_array[0],
```



```

        .close_on_exec = (fd_set *)&init_files.close_on_exec_init,
        .open_fds = (fd_set *)&init_files.open_fds_init,
    },
    .file_lock = __SPIN_LOCK_UNLOCKED(init_task.file_lock),
};

```

init_files.fdt和init_files.fdtab.fdt都分别指向了自己已有的成员变量，并以此作为一个默认值。后面的进程都是从init进程fork出来的。fork的时候会调用dup_fd，而在dup_fd中其代码结构如下：

```

newf = kmem_cache_alloc(files_cachep, GFP_KERNEL);
if (!newf)
    goto out;
atomic_set(&newf->count, 1);
spin_lock_init(&newf->file_lock);
newf->next_fd = 0;
new_fdt = &newf->fdtab;
new_fdt->max_fds = NR_OPEN_DEFAULT;
new_fdt->close_on_exec = (fd_set *)&newf->close_on_exec_init;
new_fdt->open_fds = (fd_set *)&newf->open_fds_init;
new_fdt->fd = &newf->fd_array[0];
new_fdt->next = NULL;

```

初始化new_fdt，同样是为了让new_fdt和new_fdt->fd指向其本身的成员变量fdtab和fd_array。



说明 /proc/pid/status为对应pid的进程的当前运行状态，其中FDSize值即为当前进程max_fds的值。

因此，初始状态下，files_struct、fdtable和files的关系如图1-1所示。

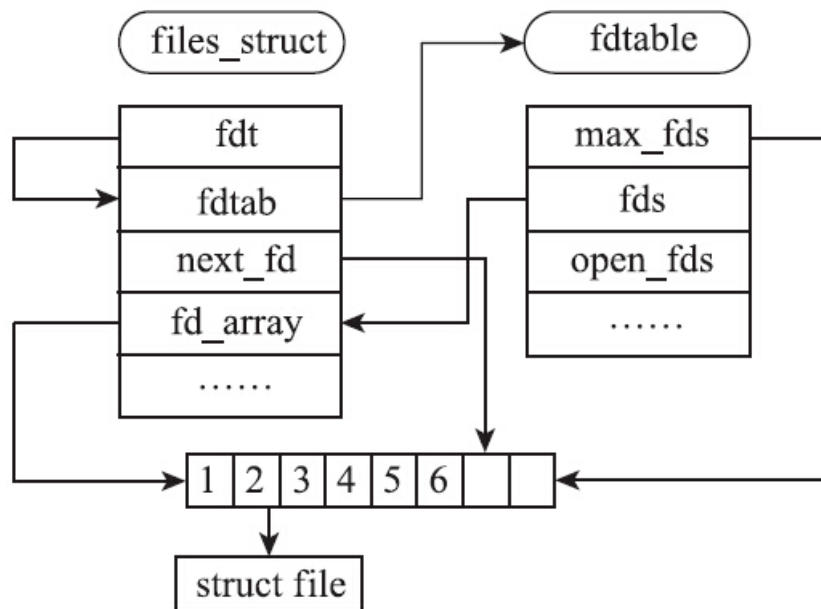


图1-1 文件表、文件描述符表及文件结构关系图

1.2 打开文件

1.2.1 open介绍

open在手册中有两个函数原型，如下所示：

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

这样的函数原型有些违背了我们的直觉。C语言是不支持函数重载的，为什么open的系统调用可以有两个这样的open原型呢？内核绝对不可能为这个功能创建两个系统调用。在Linux内核中，实际上只提供了一个系统调用，对应的是上述两个函数原型中的第二个。那么open有两个函数原型又是怎么回事呢？当我们调用open函数时，实际上调用的是glibc封装的函数，然后由glibc通过自陷指令，进行真正的系统调用。也就是说，所有的系统调用都要先经过glibc才会进入操作系统。这样的话，实际上是glibc提供了一个变参函数open来满足两个函数原型，然后通过glibc的变参函数open实现真正的系统调用来调用原型二。

可以通过一个小程序来验证我们的猜想，代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
    int fd = open("test_open.txt", O_CREAT, 0644, "test");
    close(fd);
    return 0;
}
```

在这个程序中，调用open的时候，传入了4个参数，如果open不是变参函数，就会报错，如“too many arguments to function ‘open’”。但是请看下面的编译输出：

```
[fgao@fgao-ThinkPad-R52 chapter2]#gcc -g -Wall 2_2_1_test_open.c
[fgao@fgao-ThinkPad-R52 chapter2]#
```

没有任何的警告和错误。这就证实了我们的猜想，open是glibc的一个变参函数。fcntl.h中open函数的声明也确定了这点：

```
extern int open (__const char *__file, int __oflag, ...) __nonnull ((1));
```

下面来说明一下open的参数：

·**pathname**：表示要打开的文件路径。

·**flags**: 用于指示打开文件的选项，常用的有O_RDONLY、O_WRONLY和O_RDWR。这三个选项必须有且只能有一个被指定。为什么 $O_RDWR \neq O_RDONLY | O_WRONLY$ 呢？Linux环境中，O_RDONLY被定义为0，O_WRONLY被定义为1，而O_RDWR却被定义为2。之所以有这样违反常规的设计遗留至今，就是为了兼容以前的程序。除了以上三个选项，Linux平台还支持更多的选项，APUE中对此也进行了介绍。

·**mode**: 只在创建文件时需要，用于指定所创建文件的权限位（还要受到umask环境变量的影响）。

1.2.2 更多选项

除了常用的几个打开文件的选项，APUE还介绍了一些常用的POSIX定义的选项。下面列出了Linux平台支持的大部分选项：

·**O_APPEND**：每次进行写操作时，内核都会先定位到文件尾，再执行写操作。

·**O_ASYNC**：使用异步I/O模式。

·**O_CLOEXEC**：在打开文件的时候，就为文件描述符设置FD_CLOEXEC标志。这是一个新的选项，用于解决在多线程下fork与用fcntl设置FD_CLOEXEC的竞争问题。某些应用使用fork来执行第三方的业务，为了避免泄露已打开文件的内容，那些文件会设置FD_CLOEXEC标志。但是fork与fcntl是两次调用，在多线程下，可能会在fcntl调用前，就已经fork出子进程了，从而导致该文件句柄暴露给子进程。关于O_CLOEXEC的用途，将会在第4章详细讲解。

·**O_CREAT**：当文件不存在时，就创建文件。

·**O_DIRECT**：对该文件进行直接I/O，不使用VFS Cache。

·**O_DIRECTORY**：要求打开的路径必须是目录。

·**O_EXCL**：该标志用于确保是此次调用创建的文件，需要与O_CREAT同时使用；当文件已经存在时，open函数会返回失败。

·**O_LARGEFILE**：表明文件为大文件。

·**O_NOATIME**：读取文件时，不更新文件最后的访问时间。

·**O_NONBLOCK**、**O_NDELAY**：将该文件描述符设置为非阻塞的（默认都是阻塞的）。

·**O_SYNC**：设置为I/O同步模式，每次进行写操作时都会将数据同步到磁盘，然后write才能返回。

·**O_TRUNC**：在打开文件的时候，将文件长度截断为0，需要与O_RDWR或O_WRONLY同时使用。在写文件时，如果是作为新文件重新写入，一定要使用O_TRUNC标志，否则可能会造成旧内容依然存在于文件中的错误，如生成配置文件、pid文件等——在第2章中，我会例举一个未使用截断标志而导致问题的示例代码。



注意 并不是所有的文件系统都支持以上选项。

1.2.3 open源码跟踪

我们经常这样描述“打开一个文件”，那么这个所谓的“打开”，究竟“打开”了什么？内核在这个过程中，又做了哪些事情呢？这一切将通过分析内核源码来得到答案。

跟踪内核open源码open->do_sys_open，代码如下：

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
    struct open_flags op;
    /* flags为用户层传递的参数，内核会对
```

flags进行合法性检查，并根据

mode生成新的

flags值赋给

```
    lookup */
    int lookup = build_open_flags(flags, mode, &op);
    /* 将用户空间的文件名参数复制到内核空间
```

```
*/
    char *tmp = getname(filename);
    int fd = PTR_ERR(tmp);
    if (!IS_ERR(tmp)) {
        /* 未出错则申请新的文件描述符
```

```
*/
        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
            /* 申请新的文件管理结构
```

```
file */
        struct file *f = do_filp_open(dfd, tmp, &op, lookup);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
            fd = PTR_ERR(f);
        } else {
            /* 产生文件打开的通知事件
```

```
*/
            fsnotify_open(f);
            /* 将文件描述符
```

fd与文件管理结构

file对应起来，即安装

```
*/
        fd_install(fd, f);
    }
    putname(tmp);
}
return fd;
}
```

从do_sys_open可以看出，打开文件时，内核主要消耗了两种资源：文件描述符与内核管理文件结构file。

1.2.4 如何选择文件描述符

根据POSIX标准，当获取一个新的文件描述符时，要返回最低的未使用的文件描述符。Linux是如何实现这一标准的呢？

在Linux中，通过do_sys_open->get_unused_fd_flags->alloc_fd(0, (flags))来选择文件描述符，代码如下：

```
int alloc_fd(unsigned start, unsigned flags)
{
    struct files_struct *files = current->files;
    unsigned int fd;
    int error;
    struct fdtable *fdt;
    /* files为进程的文件表，下面需要更改文件表，所以需要先锁文件表
```

```
    */
    spin_lock(&files->file_lock);
repeat:
    /* 得到文件描述符表
```

```
    */
    fdt = files_fdt(files);
    /* 从
```

start开始，查找未用的文件描述符。在打开文件时，

start为

```
0 */
    fd = start;
    /* files->next_fd为上一次成功找到的
```

fd的下一个描述符。使用

next_fd，可以快速找到未用的文

件描述符；

```
    */
    if (fd < files->next_fd)
        fd = files->next_fd;
    /*
    当小于当前文件表支持的最大文件描述符个数时，利用位图找到未用的文件描述符。
```

如果大于

max_fds怎么办呢？如果大于当前支持的最大文件描述符，那它肯定是未

用的，就不需要用位图来确认了。

```
*/
if (fd < fdt->max_fds)
    fd = find_next_zero_bit(fdt->open_fds->fds_bits,
        fdt->max_fds, fd);
/* expand_files用于在必要时扩展文件表。何时是必要的时候呢？比如当前文件描述符已经超过了当
```

前文件表支持的最大值的时候。

```
*/
error = expand_files(files, fd);
if (error < 0)
    goto out;
/*
 * If we needed to expand the fs array we
 * might have blocked - try again.
 */
if (error)
    goto repeat;
/* 只有在
```

start小于

next_fd时，才需要更新

next_fd，以尽量保证文件描述符的连续性。

```
*/
if (start <= files->next_fd)
    files->next_fd = fd + 1;
/* 将打开文件位图
```

open_fds对应

fd的位置置位

```
*/
FD_SET(fd, fdt->open_fds);
/* 根据
```

flags是否设置了

O_CLOEXEC，设置或清除


```
fdt->close_on_exec */
if (flags & O_CLOEXEC)
    FD_SET(fd, fdt->close_on_exec);
else
    FD_CLR(fd, fdt->close_on_exec);
error = fd;
#endif
/* Sanity check */
if (rcu_dereference_raw(fdt->fd[fd]) != NULL) {
    printk(KERN_WARNING "alloc fd: slot %d not NULL!\n", fd);
    rcu_assign_pointer(fdt->fd[fd], NULL);
}
#endif
out:
    spin_unlock(&files->file_lock);
    return error;
}
```

1.2.5 文件描述符fd与文件管理结构file

前文已经说过，内核使用fd_install将文件管理结构file与fd组合起来，具体操作请看如下代码：

```
void fd_install(unsigned int fd, struct file *file)
{
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    spin_lock(&files->file_lock);
    /* 得到文件描述符表

    */
    fdt = files_fdtable(files);
    BUG_ON(fdt->fd[fd] != NULL);
    /*
    将文件描述符表中的
```

file类型的指针数组中对应

fd的项指向

file。

这样文件描述符

fd与

file就建立了对应关系

```
    */
    rcu_assign_pointer(fdt->fd[fd], file);
    spin_unlock(&files->file_lock);
}
```

当用户使用fd与内核交互时，内核可以用fd从fdt->fd[fd]中得到内部管理文件的结构struct file。

1.3 creat简介

creat函数用于创建一个新文件，其等价于

open (pathname, O_WRONLY|O_CREAT|O_TRUNC, mode)。APUE介绍了引入creat的原因：

由于历史原因，早期的Unix版本中，open的第二个参数只能是0、1或者2。这样就没有办法打开一个不存在的文件。因此，一个独立系统调用creat被引入，用于创建新文件。现在的open函数，通过使用O_CREAT和O_TRUNC选项，可以实现creat的功能，因此creat已经不是必要的了。

内核creat的实现代码如下所示：

```
SYSCALL_DEFINE2(creat, const char __user *, pathname, int, mode)
{
    return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
}
```

这样就确定了creat无非是open的一种封装实现。

1.4 关闭文件

1.4.1 close介绍

`close`用于关闭文件描述符。而文件描述符可以是普通文件，也可以是设备，还可以是`socket`。在关闭时，VFS会根据不同的文件类型，执行不同的操作。

下面将通过跟踪`close`的内核源码来了解内核如何针对不同的文件类型执行不同的操作。

1.4.2 close源码跟踪

首先，来看一下close的源码实现，代码如下：

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
    struct file * filp;
    /* 得到当前进程的文件表

    */
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    int retval;
    spin_lock(&files->file_lock);
    /* 通过文件表，取得文件描述符表

    */
    fdt = files_fdt(files);
    /* 参数
```

fd大于文件描述符表记录的最大描述符，那么它一定是非法的描述符

```
    */
    if (fd >= fdt->max_fds)
        goto out_unlock;
    /* 利用
```

fd作为索引，得到

file结构指针

```
    */
    filp = fdt->fd[fd];
    /*
    检查
```

filp是否为

NULL。正常情况下，

filp一定不为

NULL。

```
    */
    if (!filp)
        goto out_unlock;
    /* 将对应的
```

filp置为

```
0*/
rcu_assign_pointer(fdt->fd[fd], NULL);
/* 清除
```

fd在

close_on_exec位图中的位

```
*/
FD_CLR(fd, fdt->close_on_exec);
/* 释放该
```

fd, 或者说将其置为

unused。

```
*/
__put_unused_fd(files, fd);
spin_unlock(&files->file_lock);
/* 关闭
```

file结构

```
*/
retval = filp_close(filp, files);
/* can't restart close syscall because file table entry was cleared */
if (unlikely(retval == -ERESTARTSYS ||
             retval == -ERESTARTNOINTR ||
             retval == -ERESTARTNOHAND ||
             retval == -ERESTART_RESTARTBLOCK))
    retval = -EINTR;
return retval;
out_unlock:
spin_unlock(&files->file_lock);
return -EBADF;
}
EXPORT_SYMBOL(sys_close);
```

__put_unused_fd源码如下所示:

```
static void __put_unused_fd(struct files_struct *files, unsigned int fd)
{
    /* 取得文件描述符表
```

```
*/
struct fdtable *fdt = files_fdtable(files);
/* 清除
```

fd在

open_fds位图的位

```
*/
    FD_CLR(fd, fdt->open_fds);
/* 如果
```

fd小于

next_fd, 重置

next_fd为释放的

```
fd */
    if (fd < files->next_fd)
        files->next_fd = fd;
}
```

看到这里，我们来回顾一下之前分析过的`alloc_fd`函数，就可以总结出完整的Linux文件描述符选择策略：

- Linux选择文件描述符是按从小到大的顺序进行寻找的，文件表中`next_fd`用于记录下一次开始寻找的起点。当有空闲的描述符时，即可分配。

- 当某个文件描述符关闭时，如果其小于`next_fd`，则`next_fd`就重置为这个描述符，这样下一次分配就会立刻重用这个文件描述符。

以上的策略，总结成一句话就是“Linux文件描述符策略永远选择最小的可用的文件描述符”。——这也是POSIX标准规定的。

其实我并不喜欢这样的策略。因为这样迅速地重用刚刚释放的文件描述符，容易引发难以调试和定位的bug——尽管这样的bug是应用层造成的。比如一个线程关闭了某个文件描述符，然后又创建了一个新的文件描述符，这时文件描述符就被重用了，但其值是一样的。如果有另外一个线程保存了之前的文件描述符的值，那它就会再次访问这个文件描述符。此时，如果是普通文件，就会读错或写错文件。如果是socket，就会与错误的对端通信。这样的错误发生时，可能并不会被察觉到。即使发现了错误，要找到根本原因，也非常困难。

如果不重用这个描述符呢？在文件描述符被关闭后，创建新的描述符且不使用相同的值。这样再次访问之前的描述符时，内核可以返回错误，应用层可以更早地得知错误的发生。

虽然造成这样错误的原因是应用层自己，但是如果内核可以尽早地让错误发生，对于应用开发人

员来说，会是一个福音。因为调试bug的时候，bug距离造成错误的地点越近，时间发生得越早，就越容易找到根本原因。这也是为什么释放内存以后，要将指针置为NULL的原因。

从__put_unused_fd退出后，close会接着调用filp_close，其调用路径为filp_close->fput。在fput中，会对当前文件struct file的引用计数减一并检查其值是否为0。当引用计数为0时，表示该struct file没有被其他人使用，则可以调用__fput执行真正的文件释放操作，然后调用要关闭文件所属文件系统的release函数，从而实现针对不同的文件类型来执行不同的关闭操作。

下一节让我们来看看Linux如何针对不同的文件类型，挂载不同的文件操作函数files_operations。

1.4.3 自定义files_operations

不失一般性，这里也选择socket文件系统作为示例，来说明Linux如何挂载文件系统指定的文件操作函数files_operations。

socket.c中定义了其文件操作函数file_operations，代码如下：

```
static const struct file_operations socket_file_ops = {
    .owner = THIS_MODULE,
    .llseek = no_llseek,
    .aio_read = sock_aio_read,
    .aio_write = sock_aio_write,
    .poll = sock_poll,
    .unlocked_ioctl = sock_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = compat_sock_ioctl,
#endif
    .mmap = sock_mmap,
    .open = sock_no_open, /* special open code to disallow open via /proc */
    .release = sock_close,
    .fsync = sock_fsync,
    .sendpage = sock_sendpage,
    .splice_write = generic_splice_sendpage,
    .splice_read = sock_splice_read,
};
```

函数sock_alloc_file用于申请socket文件描述符及文件管理结构file结构。它调用alloc_file来申请管理结构file，并将socket_file_ops作为参数，如下所示：

```
file = alloc_file(&path, FMODE_READ | FMODE_WRITE,
    &socket_file_ops);
```

进入alloc_file，来看看如下代码：

```
struct file *alloc_file(struct path *path, fmode_t mode,
    const struct file_operations *fop)
{
    struct file *file;
    /* 申请一个 */

    file = get_empty_filp();
    if (!file)
        return NULL;
    file->f_path = *path;
    file->f_mapping = path->dentry->d_inode->i_mapping;
    file->f_mode = mode;
    /* 将自定义的文件操作函数赋给 */

    file->f_op =
        file->f_op = fop;
    .....

}
```

在初始化file结构的时候，socket文件系统将其自定义的文件操作赋给了file->f_op，从而实现了在

VFS中可以调用socket文件系统自定义的操作。

1.4.4 遗忘close造成的问题

我们只需要关注close文件的时候内核做了哪些事情，就可以确定遗忘close会带来什么样的后果，如下：

- 文件描述符始终没有被释放。
- 用于文件管理的某些内存结构没有被释放。

对于普通进程来说，即使应用忘记了关闭文件，当进程退出时，Linux内核也会自动关闭文件，释放内存（详细过程见后文）。但是对于一个常驻进程来说，问题就变得严重了。

先看第一种情况，如果文件描述符没有被释放，那么再次申请新的描述符时，就不得不扩展当前的文件表了，代码如下：

```
int expand_files(struct files_struct *files, int nr)
{
    struct fdtable *fdt;
    fdt = files_fdtable(files);
    /*
     * N.B. For clone tasks sharing a files structure, this test
     * will limit the total number of files that can be opened.
     */
    if (nr >= rlimit(RLIMIT_NOFILE))
        return -EMFILE;
    /* Do we need to expand? */
    if (nr < fdt->max_fds)
        return 0;
    /* Can we expand? */
    if (nr >= sysctl_nr_open)
        return -EMFILE;
    /* All good, so we try */
    return expand_fdtable(files, nr);
}
```

从上面的代码可以看出，在扩展文件表的时候，会检查打开文件的个数是否超出系统的限制。如果文件描述符始终不释放，其个数迟早会到达上限，并返回EMFILE错误（表示Too many open files（POSIX.1））。

再看第二种情况，即文件管理的某些内存结构没有被释放。仍然是查看打开文件的代码，代码如下其中，get_empty_filp用于获得空闲的file结构。

```
struct file *get_empty_filp(void)
{
    const struct cred *cred = current_cred();
    static long old_max;
    struct file *f;
    /*
     * Privileged users can go above max_files
     */
    /* 这里对打开文件的个数进行检查，非特权用户不能超过系统的限制 */

    /*
     * if (get_nr_files() >= files_stat.max_files && !capable(CAP_SYS_ADMIN)) {
     */
    /* 再次检查 */
```

per cpu的文件个数的总和,

为什么要做两次检查呢。后文会详细介绍

```
*/
if (percpu_counter_sum_positive(&nr_files) >= files_stat.max_files)
    goto over;
}
/* 未到达上限, 申请一个新的
```

file结构

```
*/
f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
if (f == NULL)
    goto fail;
/* 增加
```

file结构计数

```
*/
percpu_counter_inc(&nr_files);
f->f_cred = get_cred(cred);
if (security_file_alloc(f))
    goto fail_sec;
INIT_LIST_HEAD(&f->u.fu_list);
atomic_long_set(&f->f_count, 1);
rwlock_init(&f->f_owner.lock);
spin_lock_init(&f->f_lock);
eventpoll_init_file(f);
/* f->f_version: 0 */
return f;
over:
/* 用完了
```

file配额, 打印

log报错

```
*/
/* Ran out of filps - report that */
if (get_nr_files() > old_max) {
    pr_info("VFS: file-max limit %lu reached\n", get_max_files());
    old_max = get_nr_files();
}
goto fail;
fail_sec:
file_free(f);
fail:
return NULL;
}
```

下面来说说为什么上面的代码要做两次检查——这也是我们学习内核代码的好处之一, 可以学到很多的编程技巧和设计思路。

对于file的个数，Linux内核使用两种方式来计数。一是使用全局变量，另外一个使用percpu变量。更新全局变量时，为了避免竞争，不得不使用锁，所以Linux使用了一种折中的解决方案。当percpu变量的个数变化不超过正负percpu_counter_batch（默认为32）的范围时，就不更新全局变量。这样就减少了对全局变量的更新，可是也造成了全局变量的值不准确的问题。于是在全局变量的file个数超过限制时，会再对所有的percpu变量求和，再次与系统的限制相比较。想了解这个计数手段的详细信息，可以阅读percpu_counter_add的相关代码。

1.4.5 如何查找文件资源泄漏

在前面的小节中，我们看到了常驻进程忘记关闭文件的危害。可是，软件不可能不出现bug，如果常驻进程程序真的出现了这样的问题，如何才能快速找到根本原因呢？通过审查打开文件的代码？时间长效率低。那是否还有其他办法呢？下面我们来介绍一种能快速查找文件资源泄漏的方法。

首先，创建一个“错误”的程序，代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    int cnt = 0;
    while (1) {
        char name[64];
        snprintf(name, sizeof(name), "%d.txt", cnt);
        int fd = creat(name, 644);
        sleep(10);
        ++cnt;
    }
    return 0;
}
```

在这段代码的循环过程中，打开了一个文件，但是一直没有被关闭，以此来模拟服务程序的文件资源泄漏，然后让程序运行一段时间：

```
[fgao@fgao chapter1]#./hold_file &
[1] 3000
```

接下来请出利器lsdf，查看相关信息，如下所示：

```
[fgao@fgao chapter1]#lsdf -p 3000
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
a.out    3000 fgao   cwd   DIR  253,2    4096 1321995 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1
a.out    3000 fgao   rtd   DIR  253,1    4096      2 /
a.out    3000 fgao   txt   REG  253,2    6115 1308841 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/a.out
a.out    3000 fgao   mem   REG  253,1   157200 1443950 /lib/ld-2.14.90.so
a.out    3000 fgao   mem   REG  253,1   2012656 1443951 /lib/libc-2.14.90.so
a.out    3000 fgao    0u   CHR 136,3      0t0      6 /dev/pts/3
a.out    3000 fgao    1u   CHR 136,3      0t0      6 /dev/pts/3
a.out    3000 fgao    2u   CHR 136,3      0t0      6 /dev/pts/3
a.out    3000 fgao    3w   REG  253,2      0 1309088 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/0.txt
a.out    3000 fgao    4w   REG  253,2      0 1312921 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/1.txt
a.out    3000 fgao    5w   REG  253,2      0 1327890 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/2.txt
a.out    3000 fgao    6w   REG  253,2      0 1327891 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/3.txt
a.out    3000 fgao    7w   REG  253,2      0 1327892 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/4.txt
a.out    3000 fgao    8w   REG  253,2      0 1327893 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/5.txt
a.out    3000 fgao    9w   REG  253,2      0 1327894 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/6.txt
```

从lsdf的输出结果可以清晰地看出，hold_file打开的哪些文件没有被关闭。其实从/proc/3000/fd中也可以得到类似的结果。但是lsdf拥有更多的选项和功能（如指定某个目录），可以应对更复杂的情况。具体细节就需要读者自行阅读lsdf的说明文档了。

1.5 文件偏移

文件偏移是基于某个打开文件来说的，一般情况下，读写操作都会从当前的偏移位置开始读写（所以`read`和`write`都没有显式地传入偏移量），并且在读写结束后更新偏移量。

1.5.1 lseek简介

lseek的原型如下：

```
off_t lseek(int fd, off_t offset, int whence);
```

该函数用于将fd的文件偏移量设置为以whence为起点，偏移为offset的位置。其中whence可以为三个值：SEEK_SET、SEEK_CUR和SEEK_END，分别表示为“文件的起始位置”、“文件的当前位置”和“文件的末尾”，而offset的取值正负均可。lseek执行成功后，会返回新的文件偏移量。

在Linux 3.1以后，Linux又增加了两个新的值：SEEK_DATA和SEEK_HOLE，分别用于查找文件中的数据 and 空洞。

1.5.2 小心lseek的返回值

对于Linux中的大部分系统调用来说，如果返回值是负数，那它一般都是错误的，但是对于lseek来说这条规则不适用。且看lseek的返回值说明：



注意 当lseek执行成功时，它会返回最终以文件起始位置为起点的偏移位置。如果出错，则返回-1，同时errno被设置为对应的错误值。

也就是说，一般情况下，对于普通文件来说，lseek都是返回非负的整数，但是对于某些设备文件来说，是允许返回负的偏移量。因此要想判断lseek是否真正出错，必须在调用lseek前将errno重置为0，然后再调用lseek，同时检查返回值是否为-1及errno的值。只有当两个同时成立时，才表明lseek真正出错了。

因为这里的文件偏移都是内核的概念，所以lseek并不会引起任何真正的I/O操作。

1.5.3 lseek源码分析

lseek的源码位于read_write.c中，如下：

```
SYSCALL_DEFINE3(lseek, unsigned int, fd, off_t, offset, unsigned int, origin)
{
    off_t retval;
    struct file * file;
    int fput_needed;
    retval = -EBADF;
    /* 根据
```

fd得到

file指针

```
*/
    file = fget_light(fd, &fput_needed);
    if (!file)
        goto bad;
    retval = -EINVAL;
    /* 对初始位置进行检查，目前
```

linux内核支持的初始位置有

1.5.1节中提到的五个值

```
*/
    if (origin <= SEEK_MAX) {
        loff_t res = vfs_llseek(file, offset, origin);
        /* 下面这段代码，先使用
```

res来给

retval赋值，然后再次判断

res
是否与

retval相等。为什么会有这样的逻辑呢？什么时候两者会不相等呢？

只有在

retval与

res的位数不相等的情况下。

retval的类型是

```
off_t->__kernel_off_t->long;
```

而

res的类型是

```
loff_t->__kernel_off_t->long long;  
在
```

32位机上, 前者是

32位, 而后者是

64位。当

res的值超过了

retval
的范围时, 两者将会不等。即实际偏移量超过了

long类型的表示范围。

```
    */  
    retval = res;  
    if (res != (loff_t)retval)  
        retval = -EOVERFLOW;    /* LFS: should only happen on 32 bit platforms */  
}  
fput_light(file, fput_needed);  
bad:  
    return retval;  
}
```

然后进入vfs_llseek, 代码如下:

```
loff_t vfs_llseek(struct file *file, loff_t offset, int origin)  
{  
    loff_t (*fn)(struct file *, loff_t, int);  
    /* 默认的
```

lseek操作是

no_llseek, 当

file没有对应的

llseek实现时, 就

会调用

no_llseek, 并返回

-ESPIPE错误

```
*/
fn = no_llseek;
if (file->f_mode & FMODE_LSEEK) {
    if (file->f_op && file->f_op->llseek)
        fn = file->f_op->llseek;
}
return fn(file, offset, origin);
}
```

当file支持llseek操作时, 就会调用具体的llseek函数。在此, 选择default_llseek作为实例, 代码如下:

```
loff_t default_llseek(struct file *file, loff_t offset, int origin)
{
    struct inode *inode = file->f_path.dentry->d_inode;
    loff_t retval;
    mutex_lock(&inode->i_mutex);
    switch (origin) {
        case SEEK_END:
            /* 最终偏移等于文件的大小加上指定的偏移量

            */
            offset += i_size_read(inode);
            break;
        case SEEK_CUR:
            /* offset为
```

0时, 并不改变当前的偏移量, 而是直接返回当前偏移量

```
*/
        if (offset == 0) {
            retval = file->f_pos;
            goto out;
        }
        /* 若
```

offset不为

0,

则最终偏移等于指定偏移加上当前偏移

```
*/
    offset += file->f_pos;
    break;
case SEEK_DATA:
    /*
     * In the generic case the entire file is data, so as
     * long as offset isn't at the end of the file then the
     * offset is data.
     */
    /* 如注释所言，对于一般文件，只要指定偏移不超过文件大小，那么指
```

定偏移的位置就是数据位置

```
*/
    if (offset >= inode->i_size) {
        retval = -ENXIO;
        goto out;
    }
    break;
case SEEK_HOLE:
    /*
     * There is a virtual hole at the end of the file, so
     * as long as offset isn't i_size or larger, return
     * i_size.
     */
    /* 只要指定偏移不超过文件大小，那么下一个空洞位置就是文件的末尾
```

```
*/
    if (offset >= inode->i_size) {
        retval = -ENXIO;
        goto out;
    }
    offset = inode->i_size;
    break;
}
retval = -EINVAL;
/* 对于一般文件来说，最终的
```

offset必须大于或等于

0，或者该文件的模式要求只能产生无符号的偏移量。否则就会报错

```
*/
if (offset >= 0 || unsigned_offsets(file)) {
    /* 当最终偏移不等于当前位置时，则更新文件的当前位置
```

```
*/
    if (offset != file->f_pos) {
        file->f_pos = offset;
        file->f_version = 0;
    }
    retval = offset;
}
out:
mutex_unlock(&inode->i_mutex);
return retval;
}
```

1.6 读取文件

Linux中读取文件操作时，最常用的就是read函数，其原型如下：

```
ssize_t read(int fd, void *buf, size_t count);
```

read尝试从fd中读取count个字节到buf中，并返回成功读取的字节数，同时将文件偏移向前移动相同的字节数。返回0的时候则表示已经到了“文件尾”。read还有可能读取比count小的字节数。

使用read进行数据读取时，要注意正确地处理错误，就是说read返回-1时，如果errno为EAGAIN、EWOULDBLOCK或EINTR，一般情况下都不能将其视为错误。因为前两者是由于当前fd为非阻塞且没有可读数据时返回的，后者是由于read被信号中断所造成的。这两种情况基本上都可以视为正常情况。

1.6.1 read源码跟踪

先来看看read的源码，代码如下：

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;
    /* 通过文件描述符

fd得到管理结构

file */
    file = fget_light(fd, &fput_needed);
    if (file) {
        /* 得到文件的当前偏移量

*/
        loff_t pos = file_pos_read(file);
        /* 利用

vfs进行真正的

read */
        ret = vfs_read(file, buf, count, &pos);
        /* 更新文件偏移量

*/
        file_pos_write(file, pos);
        /* 归还管理结构

file, 如有必要, 就进行引用计数操作

*/
        fput_light(file, fput_needed);
    }
    return ret;
}
```

再进入vfs_read，代码如下：

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    /* 检查文件是否为读取打开

*/
    if (!(file->f_mode & FMODE_READ))
        return -EBADF;
    /* 检查文件是否支持读取操作
```

```

*/
if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
/* 检查用户传递的参数

```

buf的地址是否可写

```

*/
if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
    return -EFAULT;
/* 检查要读取的文件范围实际可读取的字节数

```

```

*/
ret = rw_verify_area(READ, file, pos, count);
if (ret >= 0) {
    /* 根据上面的结构, 调整要读取的字节数

```

```

*/
    count = ret;
    /*
    如果定义

```

read操作, 则执行定义的

read操作

如果没有定义

read操作, 则调用

do_sync_read-其利用异步

aio_read来完成同步的

read操作。

```

*/
if (file->f_op->read)
    ret = file->f_op->read(file, buf, count, pos);
else
    ret = do_sync_read(file, buf, count, pos);
if (ret > 0) {
    /* 读取了一定的字节数,

```

进行通知操作

```

*/
    fsnotify_access(file);

```



```
/* 增加进程读取字节的统计计数

*/
    add_rchar(current, ret);
}
/* 增加进程系统调用的统计计数

*/
    inc_syscr(current);
}
return ret;
}
```

上面的代码为read公共部分的源码分析，具体的读取动作是由实际的文件系统决定的。

1.6.2 部分读取

前文中介绍`read`可以返回比指定`count`少的字节数，那么什么时候会发生这种情况呢？最直接的想法是在`fd`中没有指定`count`大小的数据时。但这种情况下，系统是不是也可以阻塞到满足`count`个字节的数据呢？那么内核到底采取的是哪种策略呢？

让我们来看看`socket`文件系统中`UDP`协议的`read`实现：`socket`文件系统只定义了`aio_read`操作，没有定义普通的`read`函数。根据前文，在这种情况下`do_sync_read`会利用`aio_read`实现同步读操作。

其调用链为`sock_aio_read->do_sock_read->__sock_recvmsg->__sock_recvmsg_nose->udp_recvmsg`，代码如下所示：

```
int udp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
               size_t len, int noblock, int flags, int *addr_len)
.....

    ulen = skb->len - sizeof(struct udphdr);
    copied = len;
    if (copied > ulen)
        copied = ulen;
.....
```

当`UDP`报文的数据长度小于参数`len`时，就会只复制真正的数据长度，那么对于`read`操作来说，返回的读取字节数自然就小于参数`count`了。

看到这里，是否已经得到本小节开头部分问题的答案了呢？当`fd`中的数据不够`count`大小时，`read`会返回当前可以读取的字节数？很可惜，答案是否定的。这种行为完全由具体实现来决定。即使同为`socket`文件系统，`TCP`套接字的读取操作也会与`UDP`不同。当`TCP`的`fd`的数据不足时，`read`操作极可能会阻塞，而不是直接返回。注：`TCP`是否阻塞，取决于当前缓存区可用数据多少，要读取的字节数，以及套接字设置的接收低水位大小。

因此在调用`read`的时候，只能根据`read`接口的说明，小心处理所有的情况，而不能主观臆测内核的实现。比如本文中的部分读取情况，阻塞和直接返回两种策略同时存在。

1.7 写入文件

Linux中写入文件操作，最常用的就是write函数，其原型如下：

```
ssize_t write(int fd, const void *buf, size_t count);
```

write尝试从buf指向的地址，写入count个字节到文件描述符fd中，并返回成功写入的字节数，同时将文件偏移向前移动相同的字节数。write有可能写入比指定count少的字节数。

1.7.1 write源码跟踪

write的源码与read的很相似，位于read_write.c中，代码如下：

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                 size_t, count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;
    /* 得到

file管理结构指针

    */
    file = fget_light(fd, &fput_needed);
    if (file) {
        /* 得到当前的文件偏移

    */
        loff_t pos = file_pos_read(file);
        /* 利用

VFS写入

    */
        ret = vfs_write(file, buf, count, &pos);
        /* 更新文件偏移量

    */
        file_pos_write(file, pos);
        /* 释放文件管理指针

file */
        fput_light(file, fput_needed);
    }
    return ret;
}
```

进入vfs_write，代码如下：

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    /* 检查文件是否为写入打开

    */
    if (!(file->f_mode & FMODE_WRITE))
        return -EBADF;
    /* 检查文件是否支持打开操作

    */
    if (!file->f_op || (!file->f_op->write && !file->f_op->aio_write))
        return -EINVAL;
```

/* 检查用户给定的地址范围是否可读取

```
*/  
if (unlikely(!access_ok(VVERIFY_READ, buf, count)))  
    return -EFAULT;  
/*  
验证文件从
```

pos起始是否可以写入

count个字节数

并返回可以写入的字节数

```
*/  
ret = rw_verify_area(WRITE, file, pos, count);  
if (ret >= 0) {  
    /* 更新写入字节数
```

```
*/  
    count = ret;  
    /*  
    如果定义
```

write操作, 则执行定义的

write操作

如果没有定义

write操作, 则调用

do_sync_write-其利用异步

aio_write来完成同步的

write操作

```
*/  
if (file->f_op->write)  
    ret = file->f_op->write(file, buf, count, pos);  
else  
    ret = do_sync_write(file, buf, count, pos);  
if (ret > 0) {  
    /* 写入了一定的字节数,
```

进行通知操作

```
*/
    fsnotify_modify(file);
    /* 增加进程读取字节的统计计数

*/
    add_wchar(current, ret);
    }
    /* 增加进程系统调用的统计计数

*/
    inc_syscw(current);
    }
    return ret;
}
```

write同样有部分写入的情况，这个与read类似，都是由具体实现来决定的。在此就不再深入探讨write的部分写入的情况了。

1.7.2 追加写的实现

前面说过，文件的读写操作都是从当前文件的偏移处开始的。这个文件偏移量保存在文件表中，而每个进程都有一个文件表。那么当多个进程同时写一个文件时，即使对write进行了锁保护，在进行串行写操作时，文件依然不可避免地会被写乱。根本原因就在于文件偏移量是进程级别的。

当使用O_APPEND以追加的形式来打开文件时，每次写操作都会先定位到文件末尾，然后再执行写操作。

Linux下大多数文件系统都是调用generic_file_aio_write来实现写操作的。在generic_file_aio_write中，有如下代码：

```
mutex_lock(&inode->i_mutex);
blk_start_plug(&plug);
ret = __generic_file_aio_write(iocb, iov, nr_segs, &iocb->ki_pos);
mutex_unlock(&inode->i_mutex);
```

这里有一个关键的语句，就是使用mutex_lock对该文件对应的inode进行保护，然后调用__generic_file_aio_write->generic_write_check。其部分代码如下：

```
if (file->f_flags & O_APPEND)
    *pos = i_size_read(inode);
```

上面的代码中，如果发现文件是以追加方式打开的，则将从inode中读取到的最新文件大小作为偏移量，然后通过__generic_file_aio_write再进行写操作，这样就能保证写操作是在文件末尾追加的。

1.8 文件的原子读写

使用O_APPEND可以实现在文件的末尾原子追加新数据，Linux还提供pread和pwrite从指定偏移位置读取或写入数据。

它们的实现很简单，代码如下：

```
SYSCALL_DEFINE(pread64)(unsigned int fd, char __user *buf,
                        size_t count, loff_t pos)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;
    if (pos < 0)
        return -EINVAL;
    file = fget_light(fd, &fput_needed);
    if (file) {
        ret = -ESPIPE;
        if (file->f_mode & FMODE_READ)
            ret = vfs_read(file, buf, count, &pos);
        fput_light(file, fput_needed);
    }
    return ret;
}
```

看到这段代码，是不是有一种似曾相识的感觉？让我们再来回顾一下read的实现，代码如下所示。

```
/* 得到文件的当前偏移量

*/
loff_t pos = file_pos_read(file);
/* 利用

vfs进行真正的

read */
ret = vfs_read(file, buf, count, &pos);
/* 更新文件偏移量

*/
file_pos_write(file, pos);
```

这就是它与read的主要区别。pread不会从文件表中获取当前偏移，而是直接使用用户传递的偏移量，并且在读取完毕后，不会更改当前文件的偏移量。

pwrite的实现与pread类似，在此就不再重复描述了。

1.9 文件描述符的复制

Linux提供了三个复制文件描述符的系统调用，分别为：

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);
```

其中：

- `dup`会使用一个最小的未用文件描述符作为复制后的文件描述符。

- `dup2`是使用用户指定的文件描述符`newfd`来复制`oldfd`的。如果`newfd`已经是打开的文件描述符，Linux会先关闭`newfd`，然后再复制`oldfd`。

- 对于`dup3`，只有定义了`feature`宏“`_GNU_SOURCE`”才可以使用，它比`dup2`多了一个参数，可以指定标志——不过目前仅仅支持`O_CLOEXEC`标志，可在`newfd`上设置`O_CLOEXEC`标志。定义`dup3`的原因与`open`类似，可以在进行`dup`操作的同时原子地将`fd`设置为`O_CLOEXEC`，从而避免将文件内容暴露给子进程。

为什么会有`dup`、`dup2`、`dup3`这种像兄弟一样的系统调用呢？这是因为随着软件工程的日益复杂，已有的系统调用已经无法满足需求，或者存在安全隐患，这时，就需要内核针对已有问题推出新的接口。

话说在很久以前，程序员在写`daemon`服务程序时，基本上都有这样的流程：首先关闭标准输出`stdout`、标准出错`stderr`，然后进行`dup`操作，将`stdout`或`stderr`重定向。但是在多线程程序成为主流以后，由于`close`和`dup`操作不是原子的，这就造成了在某些情况下，重定向会失败。因此就引入了`dup2`将`close`和`dup`合为一个系统调用，以保证原子性，然而这依然有问题。大家可以回顾1.2.2节中对`O_CLOEXEC`的介绍。在多线程中进行`fork`操作时，`dup2`同样会有让相同的文件描述符暴露的风险，`dup3`也就随之诞生了。这三个系统调用看起来有些冗余重复，但实际上它们也是软件工程发展的结果。从这个`dup`的发展过程来看，我们也可以领会到编写健壮代码的不易。正如前文所述，对于一个现代接口，一般都会有一个`flag`标志参数，这样既可以保证兼容性，还可以通过引用新的标志来改善或纠正接口的行为。

下面先看`dup`的实现，如下所示：

```
SYSCALL_DEFINE1(dup, unsigned int, fildes)
{
    int ret = -EBADF;
    /* 必须先得到文件管理结构
```

file, 同时也是对描述符

files的检查

```
*/
struct file *file = fget_raw(files);
if (file) {
    /* 得到一个未使用的文件描述符

*/
    ret = get_unused_fd();
    if (ret >= 0) {
        /* 将文件描述符与
```

file指针关联起来

```
*/
        fd_install(ret, file);
    }
    else
        fput(file);
    }
    return ret;
}
```

然后, 再看看fd_install的实现, 代码如下所示:

```
void fd_install(unsigned int fd, struct file *file)
{
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    /* 对文件表进行保护

*/
    spin_lock(&files->file_lock);
    /* 得到文件表

*/
    fdt = files->fdtable;
    BUG_ON(fdt->fd[fd] != NULL);
    /* 让文件表中
```

fd对应的指针等于该文件关联结构

```
file */
    rcu_assign_pointer(fdt->fd[fd], file);
    spin_unlock(&files->file_lock);
}
```

在dup中调用get_unused_fd, 只是得到一个未用的文件描述符, 那么如何实现在dup接口中使用最小的未用文件描述符呢? 这就需要回顾1.4.2节中总结过的Linux文件描述符的选择策略了。

Linux总是尝试给用户最小的未用文件描述符, 所以get_unused_fd得到的文件描述符始终是最小的

可用文件描述符。

在`fd_install`中，`fd`与`file`的关联是利用`fd`来作为指针数组的索引的，从而让对应的指针指向`file`。对于`dup`来说，这意味着数组中两个指针都指向了同一个`file`。而`file`是进程中真正的管理文件的结构，文件偏移等信息都是保存在`file`中的。这就意味着，当使用`oldfd`进行读写操作时，无论是`oldfd`还是`newfd`的文件偏移都会发生变化。

再来看一下`dup2`的实现，如下所示：

```
SYSCALL_DEFINE2(dup2, unsigned int, oldfd, unsigned int, newfd)
{
    /* 如果
```

`oldfd`与

`newfd`相等，这是一种特殊的情况

```
*/
    if (unlikely(newfd == oldfd)) { /* corner case */
        struct files_struct *files = current->files;
        int retval = oldfd;
        /*
            检查
```

`oldfd`的合法性，如果是合法的

`fd`，则直接返回

`oldfd`的值；

如果是不合法的，则返回

```
EBADF
    */
    rcu_read_lock();
    if (!fcheck_files(files, oldfd))
        retval = -EBADF;
    rcu_read_unlock();
    return retval;
}
/* 如果
```

`oldfd`与

`newfd`不同，则利用

sys_dup3来实现

```
dup2 */
    return sys_dup3(oldfd, newfd, 0);
}
```

再来查看一下dup3的实现代码，如下所示：

```
SYSCALL_DEFINE3(dup3, unsigned int, oldfd, unsigned int, newfd, int, flags)
{
    int err = -EBADF;
    struct file * file, *tofree;
    struct files_struct * files = current->files;
    struct fdtable *fdt;
    /* 对标志
```

flags进行检查，支持

```
O_CLOEXEC */
    if ((flags & ~O_CLOEXEC) != 0)
        return -EINVAL;
    /* 与
```

dup2不同，当

oldfd与

newfd相同的时候，

dup3返回错误

```
*/
    if (unlikely(oldfd == newfd))
        return -EINVAL;
    spin_lock(&files->file_lock);
    /* 根据
```

newfd决定是否需要扩展文件表的大小

```
*/
    err = expand_files(files, newfd);
    /*
    检查
```

oldfd. 如果是非法的，就直接返回

不过我更倾向于先检查

oldfd后扩展文件表，如果是非法的，就不需要扩展文件表了

```
*/
file = fcheck(oldfd);
if (unlikely(!file))
    goto Ebadf;
if (unlikely(err < 0)) {
    if (err == -EMFILE)
        goto Ebadf;
    goto out_unlock;
}
err = -EBUSY;
/* 得到文件表
```

```
*/
fdt = files_fdttable(files);
/* 通过
```

newfd得到对应的

file结构

```
*/
tofree = fdt->fd[newfd];
/*
tofree是
```

NULL, 但是

newfd已经分配的情况

```
*/
if (!tofree && FD_ISSET(newfd, fdt->open_fds))
    goto out_unlock;
/* 增加
```

file的引用计数

```
*/
get_file(file);
/* 将文件表
```

newfd对应的指针指向

```
file */
rcu_assign_pointer(fdt->fd[newfd], file);
/*
将
```

newfd加到打开文件的位图中

如果

newfd已经是一个合法的

fd, 重复设置位图则没有影响;

如果

newfd没有打开, 则必须将其加入位图中

那么为什么不对

newfd进行检查呢? 因为检查比设置位图更消耗

```
CPU
    */
    FD_SET(newfd, fdt->open_fds);
    /*
    如果
```

flags设置了

O_CLOEXEC, 则将

newfd加到

close_on_exec位图;

如果没有设置, 则清除

close_on_exec位图中对应的位

```
*/
if (flags & O_CLOEXEC)
    FD_SET(newfd, fdt->close_on_exec);
else
    FD_CLR(newfd, fdt->close_on_exec);
spin_unlock(&files->file_lock);
/* 如果
```

tofree不为空, 则需要关闭

newfd之前的文件

```
*/
    if (tofree)
        filp_close(tofree, files);
    return newfd;
Ebadf:
    err = -EBADF;
out_unlock:
    spin_unlock(&files->file_lock);
    return err;
}
```

1.10 文件数据的同步

为了提高性能，操作系统会对文件的I/O操作进行缓存处理。对于读操作，如果要读取的内容已经存在于文件缓存中，就直接读取文件缓存。对于写操作，会先将修改提交到文件缓存中，在合适的时机或者过一段时间后，操作系统才会将改动提交到磁盘上。

Linux提供了三个同步接口：

```
void sync(void);
int fsync(int fd);
int fdatasync(int fd);
```

APUE上说sync只是让所有修改过的缓存进入提交队列，并不用等待这个工作完成。Linux手册上则表示从1.3.20版本开始，Linux就会一直等待，直到提交工作完成。

实际情况到底是怎样的呢，让代码告诉我们真相，具体如下：

```
SYSCALL_DEFINE0(sync)
{
    /* 唤醒后台内核线程，将“脏”缓存冲刷到磁盘上

    */
    wakeup_flusher_threads(0, WB_REASON_SYNC);
    /*
    为什么要调用两次
```

sync_filesystems呢？

这是一种编程技巧，第一次

sync_filesystems(0)，参数

0表示不等待，可以

迅速地将没有上锁的

inode同步。第二次

sync_filesystems(1)，参数

1表示等待。

对于上锁的

inode会等待到解锁，再执行同步，这样可以提高性能。因为第一次操作

中，上锁的

inode很可能在第一次操作结束后，就已经解锁，这样就避免了等待

```
*/
sync_filesystems(0);
sync_filesystems(1);
/*
如果是
```

laptop模式，那么因为此处刚刚做完同步，因此可以停掉后台同步定时器

```
*/
if (unlikely(laptop_mode))
    laptop_sync_completion();
return 0;
}
```

再看一下sync_filesystems->iterate_supers->sync_one_sb->__sync_filesystem，代码如下：

```
static int __sync_filesystem(struct super_block *sb, int wait)
{
    /*
     * This should be safe, as we require bdi backing to actually
     * write out data in the first place
     */
    if (sb->s_bdi == &noop_backing_dev_info)
        return 0;
    /* 磁盘配额同步

    */
    if (sb->s_qcop && sb->s_qcop->quota_sync)
        sb->s_qcop->quota_sync(sb, -1, wait);
    /*
    如果
```

wait为

true，则一直等待直到所有的脏

inode写入磁盘

如果

wait为

false, 则启动脏

inode回写工作, 但不必等待到结束

```
*/
if (wait)
    sync_inodes_sb(sb);
else
    writeback_inodes_sb(sb, WB_REASON_SYNC);
/* 如果该文件系统定义了自己的同步操作, 则执行该操作

*/
if (sb->s_op->sync_fs)
    sb->s_op->sync_fs(sb, wait);
/* 调用
```

block设备的

flush操作, 真正地将数据写到设备上

```
*/
return __sync_blockdev(sb->s_bdev, wait);
}
```

从sync的代码实现上看, Linux的sync是阻塞调用, 这里与APUE的说明是不一样的。

下面来看看fsync与fdatasync, fsync只同步fd指定的文件, 并且直到同步完成才返回。fdatasync与fsync类似, 但是其只同步文件的实际数据内容, 和会影响后面数据操作的元数据。而fsync不仅同步数据, 还会同步所有被修改过的文件元数据, 代码如下所示:

```
SYSCALL_DEFINE1(fsync, unsigned int, fd)
{
    return do_fsync(fd, 0);
}
SYSCALL_DEFINE1(fdatasync, unsigned int, fd)
{
    return do_fsync(fd, 1);
}
```

事实上, 真正进行工作的是do_fsync, 代码如下所示:

```
static int do_fsync(unsigned int fd, int datasync)
{
    struct file *file;
    int ret = -EBADF;
    /* 得到
```

file管理结构

```
*/
file = fget(fd);
if (file) {
    /* 利用
```

vfs执行

sync操作

```
*/
    ret = vfs_fsync(file, datasync);
    fput(file);
}
return ret;
}
```

进入vfs_fsync->vfs_fsync_range，代码如下：

```
int vfs_fsync_range(struct file *file, loff_t start, loff_t end, int datasync)
{
    /* 调用具体操作系统的同步操作

*/
    if (!file->f_op || !file->f_op->fsync)
        return -EINVAL;
    return file->f_op->fsync(file, start, end, datasync);
}
```

真正执行同步操作的fsync是由具体的文件系统的操作函数file_operations决定的。下面选择一个常用的文件系统同步函数generic_file_fsync，代码如下。

```
int generic_file_fsync(struct file *file, loff_t start, loff_t end,
                      int datasync)
{
    struct inode *inode = file->f_mapping->host;
    int err;
    int ret;
    /* 同步该文件缓存中处于
```

start到

end范围内的脏页

```
*/
    err = filemap_write_and_wait_range(inode->i_mapping, start, end);
    if (err)
        return err;
    mutex_lock(&inode->i_mutex);
    /* 同步该
```

inode对应的缓存

```

*/
ret = sync_mapping_buffers(inode->i_mapping);
/* inode状态没有变化, 无需同步, 可以直接返回

*/
if (!(inode->i_state & I_DIRTY))
    goto out;
/* 如果是

```

fdatasync则仅做数据同步, 并且若该

inode没有影响任何数据方面操作的变化(比如文件长度), 则可以直接返回

```

*/
if (datasync && !(inode->i_state & I_DIRTY_DATASYNC))
    goto out;
/*
同步

```

inode的元数据

```

*/
err = sync_inode_metadata(inode, 1);
if (ret == 0)
    ret = err;
out:
mutex_unlock(&inode->i_mutex);
return ret;
}

```

从上面的代码可以看出, fdatsync的性能会优于fsync。在不需要同步所有元数据的情况下, 选择fdatsync会得到更好的性能。只有在inode被设置了I_DIRTY_DATASYNC标志时, fdatsync才需要同步inode的元数据。那么inode何时会被设置I_DIRTY_DATASYNC这个标志呢? 比如使用文件截断truncate或ftruncate时; 通过在源码中搜索I_DIRTY_DATASYNC或mark_inode_dirty时也会给inode设置该标志位。而调用mark_inode_dirty的地方就太多了, 这里就不一一列举了。



注意 sync、fsync和fdatsync只能保证Linux内核对文件的缓冲被冲刷了, 并不能保证数据被真正写到磁盘上, 因为磁盘也有自己的缓存。

1.11 文件的元数据

1.10节中我们提到了文件元数据，那么什么是文件的元数据呢？其包括文件的访问权限、上次访问的时间戳、所有者、所有组、文件大小等信息。

1.11.1 获取文件的元数据

Linux环境提供了三个获取文件信息的API:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

这三个函数都可用于得到文件的基本信息，区别在于stat得到路径path所指定的文件基本信息，fstat得到文件描述符fd指定文件的基本信息，而lstat与stat则基本相同，只有当path是一个链接文件时，lstat得到的是链接文件自己本身的基本信息而不是其指向文件的信息。

所得到的文件基本信息的结果struct stat的结构如下:

```
struct stat {
    dev_t    st_dev;        /* ID of device containing file */
    ino_t     st_ino;        /* inode number */
    mode_t    st_mode;       /* protection */
    nlink_t   st_nlink;      /* number of hard links */
    uid_t     st_uid;        /* user ID of owner */
    gid_t     st_gid;        /* group ID of owner */
    dev_t     st_rdev;       /* device ID (if special file) */
    off_t     st_size;       /* total size, in bytes */
    blksize_t st_blksize;    /* blocksize for file system I/O */
    blkcnt_t  st_blocks;     /* number of 512B blocks allocated */
    time_t    st_atime;      /* time of last access */
    time_t    st_mtime;      /* time of last modification */
    time_t    st_ctime;      /* time of last status change */
};
```

Linux的man手册对stat的各个变量做了注释，明确指出了每个变量的意义。唯一需要说明的是st_mode，其不仅仅是注释所说的“protection”，即权限管理，同时也用于表示文件类型，比如是普通文件还是目录。

1.11.2 内核如何维护文件的元数据

要搞清楚Linux如何维护文件的元数据，就需要追踪stat的实现，具体代码如下：

```
SYSCALL_DEFINE2(stat, const char __user *, filename,
                struct __old_kernel_stat __user *, statbuf)
{
    struct kstat stat;
    int error;
    /* vfs_stat用于读取文件元数据至

stat */
    error = vfs_stat(filename, &stat);
    if (error)
        return error;
    /* 这里仅是从内核的元数据结构

stat复制到用户层的数据结构

statbuf中

*/
    return cp_old_stat(&stat, statbuf);
}
```

进入vfs_stat->vfs_fstatat->vfs_getattr，代码如下：

```
int vfs_getattr(struct vfsmount *mnt, struct dentry *dentry, struct kstat *stat)
{
    struct inode *inode = dentry->d_inode;
    int retval;
    /* 对获取

inode属性操作进行安全性检查

*/
    retval = security_inode_getattr(mnt, dentry);
    if (retval)
        return retval;
    /* 如果该文件系统定义了这个

inode的自定义操作函数，就执行它

*/
    if (inode->i_op->getattr)
        return inode->i_op->getattr(mnt, dentry, stat);
    /* 如果文件系统没有定义

inode的操作函数，则执行通用的函数

*/
}
```

```
generic_fillattr(inode, stat);  
return 0;  
}
```

不失一般性，也可以通过查看generic_fillattr来进一步了解，代码如下：

```
void generic_fillattr(struct inode *inode, struct kstat *stat)  
{  
    stat->dev = inode->i_sb->s_dev;  
    stat->ino = inode->i_ino;  
    stat->mode = inode->i_mode;  
    stat->nlink = inode->i_nlink;  
    stat->uid = inode->i_uid;  
    stat->gid = inode->i_gid;  
    stat->rdev = inode->i_rdev;  
    stat->size = i_size_read(inode);  
    stat->atime = inode->i_atime;  
    stat->mtime = inode->i_mtime;  
    stat->ctime = inode->i_ctime;  
    stat->blksize = (1 << inode->i_blkbits);  
    stat->blocks = inode->i_blocks;  
}
```

从这里可以看出，所有的文件元数据均保存在inode中，而inode是Linux也是所有类Unix文件系统中的概念。这样的文件系统一般将存储区域分为两类，一类是保存文件对象的元信息数据，即inode表；另一类是真正保存文件数据内容的块，所有inode完全由文件系统来维护。但是Linux也可以挂载非类Unix的文件系统，这些文件系统本身没有inode的概念，怎么办？Linux为了让VFS有统一的处理流程和方法，就必须要求那些没有inode概念的文件系统，根据自己系统的特点——如何维护文件元数据，生成“虚拟的”inode以供Linux内核使用。

1.11.3 权限位解析

在Linux环境中，文件常见的权限位有r、w和x，分别表示可读、可写和可执行。下面重点解析三个不常用的标志位。

1.SUID权限位

当文件设置SUID权限位时，就意味着无论是谁执行这个文件，都会拥有该文件所有者的权限。`passwd`命令正是利用这个特性，来允许普通用户修改自己的密码，因为只有root用户才有修改密码文件的权限。当普通用户执行`passwd`命令时，就具有了root权限，从而可以修改自己的密码。

以修改文件属性的权限检查代码为例，`inode_change_ok`用于检查该进程是否有权限修改inode节点的属性即文件属性，示例代码如下：

```
int inode_change_ok(const struct inode *inode, struct iattr *attr)
{
    unsigned int ia_valid = attr->ia_valid;
    .....

    /* Make sure a caller can chown. */
    /* 只有在

uid和

suid都不符合条件的情况下，才会返回权限不足的错误

*/
    if ((ia_valid & ATTR_UID) &&
        (current_fsuid() != inode->i_uid ||
         attr->ia_uid != inode->i_uid) && !capable(CAP_CHOWN))
        return -EPERM;
    .....

}
```

2.SGID权限位

SGID与SUID权限位类似，当设置该权限位时，就意味着无论是谁执行该文件，都会拥有该文件所有者所在组的权限。

3.Stricky位

Stricky位只有配置在目录上才有意义。当目录配置上sticky位时，其效果是即使所有的用户都拥有写权限和执行权限，该目录下的文件也只能被root或文件所有者删除。

下面来看看内核的实现：

```
static int may_delete(struct inode *dir, struct dentry *victim, int isdir)
{
    .....

    if (check_sticky(dir, victim->d_inode) ||
        IS_APPEND(victim->d_inode) ||
        IS_IMMUTABLE(victim->d_inode) ||
        IS_SWAPFILE(victim->d_inode))
        return -EPERM;
    .....

}
```

在删除文件前，内核要调用may_delete来判断该文件是否可以被删除。在这个函数中，内核通过调用check_sticky来检查文件的sticky标志位，其代码如下：

```
static inline int check_sticky(struct inode *dir, struct inode *inode)
{
    /* 得到当前文件访问权限的

uid */
    uid_t fsuid = current_fsuid();
    /* 判断上级目录是否设置了

sticky标志位

*/
    if (!(dir->i_mode & S_ISVTX))
        return 0;
    /* 检查名称空间

*/
    if (current_user_ns() != inode_userns(inode))
        goto other_userns;
    /* 检查当前文件的

uid是否与当前用户的

uid相同

*/
    if (inode->i_uid == fsuid)
        return 0;
    /* 检查文件所处目录的
```

uid是否与当前用户的

uid相同

```
*/
    if (dir->i_uid == fsuid)
        return 0;
    /* 该文件不属于当前用户

other_userns:
    return !ns_capable(inode_userns(inode), CAP_FOWNER);
}
```

当文件所处的目录设置了sticky位，即使用户拥有了对应的权限，只要不是目录或文件的拥有者，就无法删除该文件——除非该用户拥有CAP_FOWNER能力（读者可以通过man 7 capabilities来进一步了解Linux中的capabilities。一般只有root用户才有这样的能力）。



说明 大家可以使用chmod来设置文件或目录的权限。

1.12 文件截断

1.12.1 truncate与ftruncate的简单介绍

Linux提供了两个截断文件的API:

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

两者之间的唯一区别在于，`truncate`截断的是路径`path`指定的文件，`ftruncate`截断的是`fd`引用的文件。

“截断”给人的感觉是将文件变短，即将文件大小缩短至`length`长度。实际上，`length`可以大于文件本身的大小，这时文件长度将变为`length`的大小，扩充的内容均被填充为0。需要注意的是，尽管`ftruncate`使用的是文件描述符，但是其并不会更新当前文件的偏移。

1.12.2 文件截断的内核实现

先来看看truncate的内核实现，代码如下：

```
SYSCALL_DEFINE2(truncate, const char __user *, path, long, length)
{
    return do_sys_truncate(path, length);
}
```

进入do_sys_truncate，代码如下：

```
static long do_sys_truncate(const char __user *pathname, loff_t length)
{
    struct path path;
    struct inode *inode;
    int error;
    error = -EINVAL;
    /* 长度不能为负数

*/
    if (length < 0)
        goto out;
    /* 得到路径结构

*/
    error = user_path(pathname, &path);
    if (error)
        goto out;
    inode = path.dentry->d_inode;
    error = -EISDIR;
    /* 目录不能被截断

*/
    if (S_ISDIR(inode->i_mode))
        goto dput_and_out;
    error = -EINVAL;
    /* 不是普通文件不能被截断

*/
    if (!S_ISREG(inode->i_mode))
        goto dput_and_out;
    /* 尝试获得文件系统的写权限

*/
    error = mnt_want_write(path.mnt);
    if (error)
        goto dput_and_out;
    /* 检查是否有文件写权限

*/
    error = inode_permission(inode, MAY_WRITE);
    if (error)
        goto mnt_drop_write_and_out;
    error = -EPERM;
    /* 文件设置了追加属性，则不能被截断

*/
    if (IS_APPEND(inode))
        goto mnt_drop_write_and_out;
    /* 得到
```

inode的写权限

```
*/
error = get_write_access(inode);
if (error)
    goto mnt_drop_write_and_out;
/* 查看是否与文件
```

lease锁相冲突

```
*/
error = break_lease(inode, O_WRONLY);
if (error)
    goto put_write_and_out; //
```

* 检查是否与文件锁相冲突

```
*/
```

```
error = locks_verify_truncate(inode, NULL, length);
if (!error)
    error = security_path_truncate(&path);
/* 如果没有错误, 则进行真正的截断
```

```
*/
if (!error)
    error = do_truncate(path.dentry, length, 0, NULL);
put_write_and_out:
put_write_access(inode);
mnt_drop_write_and_out:
mnt_drop_write(path.mnt);
dput_and_out:
path_put(&path);
out:
return error;
}
```

再进入do_truncate, 代码如下:

```
int do_truncate(struct dentry *dentry, loff_t length, unsigned int time_attrs,
                struct file *filp)
{
    int ret;
    struct iattr newattrs;
    if (length < 0)
        return -EINVAL;
    /* 设置要改变的属性, 对于截断来说, 最重要的是文件长度

*/
    newattrs.ia_size = length;
    newattrs.ia_valid = ATTR_SIZE | time_attrs;
    if (filp) {
        newattrs.ia_file = filp;
        newattrs.ia_valid |= ATTR_FILE;
    }
    /*
    suid权限一定会被去掉
```

同时设置

sgid和

xgrp时,

sgid权限也会被去掉

```
*/
ret = should_remove_suid(dentry);
if (ret)
    newattrs.ia_valid |= ret | ATTR_FORCE;
/* 修改
```

inode属性

```
*/
mutex_lock(&dentry->d_inode->i_mutex);
ret = notify_change(dentry, &newattrs);
mutex_unlock(&dentry->d_inode->i_mutex);
return ret;
}
```

接下来看ftruncate的实现，代码如下：

```
SYSCALL_DEFINE2(ftruncate, unsigned int, fd, unsigned long, length)
{
    /* 真正的工作函数

do_sys_ftruncate */
    long ret = do_sys_ftruncate(fd, length, 1);
    /* avoid REGPARM breakage on x86: */
    asmlinkage_protect(2, ret, fd, length);
    return ret;
}
```

最后，进入do_sys_ftruncate，代码如下：

```
static long do_sys_ftruncate(unsigned int fd, loff_t length, int small)
{
    struct inode * inode;
    struct dentry *dentry;
    struct file * file;
    int error;
    error = -EINVAL;
    /* 长度检查

*/
    if (length < 0)
        goto out;
    error = -EBADF;
    /* 从文件描述符得到
```

file指针

```
*/
file = fget(fd);
if (!file)
    goto out;
/* 如果文件是以
```

O_LARGEFILE选项打开的, 则将标志

small置为

0即假

```
*/
if (file->f_flags & O_LARGEFILE)
    small = 0;
dentry = file->f_path.dentry;
inode = dentry->d_inode;
error = -EINVAL;
/* 如果文件不是普通文件或文件不是写打开, 则报错
```

```
*/
if (!S_ISREG(inode->i_mode) || !(file->f_mode & FMODE_WRITE))
    goto out_putf;
error = -EINVAL; /* Cannot ftruncate over 2^31 bytes without large file support */
/* 如果文件不是以
```

O_LARGEFILE打开的话, 长度就不能超过

```
MAX_NON_LFS */
if (small && length > MAX_NON_LFS)
    goto out_putf;
error = -EPERM;
/* 如果是追加模式打开的, 也不能进行截断
```

```
*/
if (IS_APPEND(inode))
    goto out_putf;
/* 检查是否有锁冲突
```

```
*/
error = locks_verify_truncate(inode, file, length);
if (!error)
    error = security_path_truncate(&file->f_path);
if (!error) {
    /* 执行截断操作-前文已经分析过
```

```
*/
error = do_truncate(dentry, length, ATTR_MTIME|ATTR_CTIME, file);}
out_putf:
    fput(file);
out:
    return error;
}
```

1.12.3 为什么需要文件截断

文件截断时允许指定比原有文件长度更长的值，但更常见的是指定的长度比原有长度短，这主要用于防止文件内容混杂了旧内容的情况。下面以常见的daemon程序为例（演示一个文件因不截断而引发的bug），这种程序往往要将自己的pid写入一个pid文件中。当daemon程序启动的时候，最好是将旧的pid文件截断，然后写入新的pid，不然pid文件中可能会保存错误的pid。

假设当前的test.pid文件的内容是上一次的pid。

```
[fgao@fgao chapter1]#cat test.pid
123456
```

下面的程序是将新的pid——6789写入test.pid中。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
    int fd = open("test.pid", O_WRONLY);
    write(fd, "6789", sizeof("6789")-1);
    close(fd);
    return 0;
}
```

程序执行完毕，让我们看看test.pid的内容：

```
[fgao@fgao chapter1]#cat test.pid
678956
```

这显然不是我们所期望的结果。为了解决这个问题，我们可以在打开文件的同时，指定O_TRUNC标志。

```
int fd = open("test.pid", O_WRONLY | O_TRUNC);
```

或者使用本节介绍的截断API，代码如下：

```
truncate("test.pid", 0);
int fd = open("test.pid", O_WRONLY);
```

或者用如下代码：

```
int fd = open("test.pid", O_WRONLY);
ftruncate(fd, 0);
```

这样，就能保证旧内容不会与最新写入的内容混杂在一起。

也许有朋友会提出，在上面的例子中写入“6789”时，这样写就不会有问题了：

```
write(fd, "6789", sizeof("6789"));
```

然而结果仍然是错的，其结果为：

```
[fgao@ubuntu chapter1]#cat test.pid  
67896
```

这里列举的例子用的是文本文件，如果写入的是一个二进制文件，当不使用文件截断而导致新旧数据混杂在一起时，定位错误将更加困难。所以，在我们的日常编码中，在写入文件，如果并不需要旧数据，那么在打开文件时就要强制截断文件，来提高代码的健壮性。

第2章 标准I/O库

前面的章节介绍的是Linux的系统调用。本章将从标准I/O库开始讲解Linux环境编程中不可或缺的C库。在学习和分析标准I/O库的同时，与Linux的I/O系统调用进行比较，可以加深对两者的认识和理解。

2.1 stdin、stdout和stderr

当Linux新建一个进程时，会自动创建3个文件描述符0、1和2，分别对应标准输入、标准输出和错误输出。C库中与文件描述符对应的是文件指针，与文件描述符0、1和2类似，我们可以直接使用文件指针stdin、stdout和stderr。那么这是否意味着stdin、stdout和stderr是“自动打开”的文件指针呢？

查看C库头文件stdio.h中的源码：

```
typedef struct _IO_FILE FILE;
/* Standard streams. */
extern struct _IO_FILE *stdin;      /* Standard input stream. */
extern struct _IO_FILE *stdout;     /* Standard output stream. */
extern struct _IO_FILE *stderr;     /* Standard error output stream. */
#ifdef STDC
/* C89/C99 say they're macros. Make them happy. */
#define stdin stdin
#define stdout stdout
#define stderr stderr
#endif
```

从上面的源码可以看出，stdin、stdout和stderr确实是文件指针。而C标准要求stdin、stdout和stderr是宏定义，所以在C库的代码中又定义了同名宏。

那么stdin、stdout和stderr又是如何定义的呢？定义代码如下：

```
_IO_FILE *stdin = (FILE *) &_IO_2_1_stdin_;
_IO_FILE *stdout = (FILE *) &_IO_2_1_stdout_;
_IO_FILE *stderr = (FILE *) &_IO_2_1_stderr_;
```

继续查看_IO_2_1_stdin_等的定义，代码如下：

```
DEF_STDFILE(_IO_2_1_stdin_, 0, 0, _IO_NO_WRITES);
DEF_STDFILE(_IO_2_1_stdout_, 1, &_IO_2_1_stdin_, _IO_NO_READS);
DEF_STDFILE(_IO_2_1_stderr_, 2, &_IO_2_1_stdout_, _IO_NO_READS+_IO_UNBUFFERED);
```

DEF_STDFILE是一个宏定义，用于初始化C库中的FILE结构。这里_IO_2_1_stdin_、_IO_2_1_stdout_和_IO_2_1_stderr_这三个FILE结构分别用于文件描述符0、1和2的初始化，这样C库的文件指针就与系统的文件描述符互相关联起来了。大家注意最后的标志位，stdin是不可写的，stdout是不可读的，而stderr不仅不可读，且没有缓存。

通过上面的分析，可以得到一个结论：stdin、stdout和stderr都是FILE类型的文件指针，是由C库静态定义的，直接与文件描述符0、1和2相关联，所以应用程序可以直接使用它们。

2.2 I/O缓存引出的趣题

C库的I/O接口对文件I/O进行了封装，为了提高性能，其引入了缓存机制，共有三种缓存机制：全缓存、行缓存及无缓存。

- 全缓存一般用于访问真正的磁盘文件。C库会为文件访问申请一块内存，只有当文件内容将缓存填满或执行冲刷函数flush时，C库才会将缓存内容写入内核中。

- 行缓存一般用于访问终端。当遇到一个换行符时，就会引发真正的I/O操作。需要注意的是，C库的行缓存也是固定大小的。因此，当缓存已满，即使没有换行符时也会引发I/O操作。

- 无缓存，顾名思义，C库没有进行任何的缓存。任何C库的I/O调用都会引发实际的I/O操作。

C库提供了接口，用于修改默认的缓存行为，相关代码如下：

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

下面看一个跟C库缓存相关的趣题。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    printf("Hello ");
    if (0 == fork()) {
        printf("child\n");
        return 0;
    }
    printf("parent\n");
    return 0;
}
```

其输出结果是什么？正确的结果是：

```
Hello parent
Hello child
```

或者：

```
Hello child
Hello parent
```

之所以是这样的结果，就是因为背后的行缓存。执行printf("Hello")时，因为printf是向标准输出打印的，因此使用的是行缓存。字符串Hello没有换行符，所以并没有真正的I/O输出。当执行fork时，子进程会完全复制父进程的内存空间，因此字符串Hello也存在于子进程的行缓存中。故而最后的输出

结果中，无论是父进程还是子进程都有Hello字符串。

2.3 fopen和open标志位对比

C库的fopen用于打开文件，其内部实现必然要使用open系统调用。那么fopen的各个标志位又对应open的哪些标志位呢？请看表2-1。

表2-1 fopen标志位和open标志位对应表

fopen 标志位	open 标志位	用 途
r	O_RDONLY	以只读方式打开文件
r+	O_RDWR	以读写方式打开文件
w	O_WRONLY O_CREAT O_TRUNC	以写方式打开文件；当文件存在时，将其大小截断为 0；当文件不存在时，创建该文件
w+	O_RDWR O_CREAT O_TRUNC	以读写方式打开文件；当文件存在时，将其大小截断为 0；当文件不存在时，创建该文件

(续)

fopen 标志位	open 标志位	用 途
a	O_WRONLY O_APPEND O_CREAT	以追加写的方式打开文件，当文件不存在时，创建该文件
a+	O_RDWR O_APPEND O_CREAT	以追加读写的方式打开文件，当文件不存在时，创建该文件

表2-1是fopen常用的标志位，实际上fopen还有更多的标志位，这也是很多书籍没有涉及的，具体见表2-2。

表2-2 更多的fopen和open标志位对应

fopen 标志位	open 标志位	用 途
c	无	该文件流在 I/O 操作时不能被取消
e	O_CLOEXEC	当进程执行 exec 时，该文件流会自动关闭
m	无	该文件流通过 mmap 来打开或访问，只支持读取操作
x	O_EXCL	在创建文件时，如果文件已经存在，fopen 则会返回失败而不是打开这个文件
b	无	表示打开的文件是二进制流而不是文本流。该标志目前在 Linux 中是无用的

下面进入glibc的源码，查看函数_IO_new_file_fopen来验证上面的结论。

```
_IO_FILE *
_IO_new_file_fopen (fp, filename, mode, is32not64)
{
  _IO_FILE *fp;
  const char *filename;
  const char *mode;
  int is32not64;

  {
    int oflags = 0, omode;
    int read_write;
    int oprot = 0666;
    int i;
    _IO_FILE *result;
#ifdef _LIBC
    const char *cs;
    const char *last_recognized;
#endif
    if (_IO_file_is_open (fp))
      return 0;
    switch (*mode)
    {
      case 'r':
        omode = O_RDONLY;
        read_write = _IO_NO_WRITES;
        break;
      case 'w':
        omode = O_WRONLY;
        oflags = O_CREAT|O_TRUNC;
        read_write = _IO_NO_READS;
```

```

        break;
    case 'a':
        omode = O_WRONLY;
        oflags = O_CREAT|O_APPEND;
        read_write = _IO_NO_READS|_IO_IS_APPENDING;
        break;
    default:
        __set_errno (EINVAL);
        return NULL;
    }
#ifdef _LIBC
    last_recognized = mode;
#endif
    for (i = 1; i < 7; ++i)
    {
        switch (*++mode)
        {
            case '\0':
                break;
            case '+':
                omode = O_RDWR;
                read_write &= _IO_IS_APPENDING;
#ifdef _LIBC
                last_recognized = mode;
#endif
                continue;
            case 'x':
                oflags |= O_EXCL;
#ifdef _LIBC
                last_recognized = mode;
#endif
                continue;
            case 'b':
#ifdef _LIBC
                last_recognized = mode;
#endif
                continue;
            case 'm':
                fp->flags2 |= _IO_FLAGS2_MMAP;
                continue;
            case 'c':
                fp->flags2 |= _IO_FLAGS2_NOTCANCEL;
                continue;
            case 'e':
#ifdef O_CLOEXEC
                oflags |= O_CLOEXEC;
#endif
#ifdef _LIBC
                fp->flags2 |= _IO_FLAGS2_CLOEXEC;
                continue;
            default:
                /* Ignore. */
                continue;
        }
        break;
    }
}
result = _IO_file_open (fp, filename, omode|oflags, oprot, read_write,
                        is32not64);

```

上面的源代码非常简单，很容易理解。每个mode都是switch语句的一个case，oflags就是要传给open的标志位，这就验证了前文的结论。

2.4 fdopen与fileno

Linux提供了文件描述符，而C库又提供了文件流。在平时的工作中，有时候需要在两者之间进行切换，因此C库提供了两个API:

```
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
int fileno(FILE *stream);
```

`fdopen`用于从文件描述符`fd`生成一个文件流`FILE`，而`fileno`则用于从文件流`FILE`得到对应的文件描述符。

查看`fdopen`的实现，其基本工作是创建一个新的文件流`FILE`，并建立文件流`FILE`与描述符的对应关系。我们以`fileno`的简单实现，来了解文件流`FILE`与文件描述符`fd`的关系。——因为该函数代码较长，在此就不罗列C库的代码了。代码如下：

```
int fileno (_IO_FILE* fp)
{
    CHECK_FILE (fp, EOF);
    if (!(fp->_flags & _IO_IS_FILEBUF) || _IO_fileno (fp) < 0)
    {
        __set_errno (EBADF);
        return -1;
    }
    return _IO_fileno (fp);
}
#define _IO_fileno(FP) ((FP)->_fileno)
```

从`fileno`的实现基本上就可以得知文件流与文件描述符的对应关系。文件流`FILE`保存了文件描述符的值。当从文件流转换到文件描述符时，可以直接通过当前`FILE`保存的值`_fileno`得到`fd`。而从文件描述符转换到文件流时，C库返回的都是一个重新申请的文件流`FILE`，且这个`FILE`的`_fileno`保存了文件描述符。

因此无论是`fdopen`还是`fileno`，关闭文件时，都要使用`fclose`来关闭文件，而不是用`close`。因为只有采用此方式，`fclose`作为C库函数，才会释放文件流`FILE`占用的内存。

2.5 同时读写的痛苦

前面介绍过内核的文件描述符实现。在内核中，每一个文件描述符`fd`都对应了一个文件管理结构`struct file`——用于维护该文件描述符的信息，如偏移量等。在第1章对`read`和`write`的源码分析中，可以发现每一次系统调用的`read`和`write`成功返回后，文件的偏移量都会被更新。

因此，如果程序对同一个文件描述符进行读写操作的话，肯定会得到非期望的结果，示例代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    char buf[20];
    int ret;
    FILE *fp = fopen("./tmp.txt", "w+");
    if (!fp) {
        printf("Fail to open file\n");
        return -1;
    }
    ret = fwrite("123", sizeof("123"), 1, fp);
    printf("we write %d member\n", ret);
    memset(buf, 0, sizeof(buf));
    ret = fread(buf, 1, 1, fp);
    printf("We read %s, ret is %d\n", buf, ret);
    fwrite("456", sizeof("456"), 1, fp);
    fclose(fp);
    return 0;
}
```

上面的代码中，利用`fopen`的读写模式打开了一个文件流，先写入一个字符串“123”，然后读取一个字节，再写入一个字符串“456”。

大家想想输出结果会是什么呢？`fread`读取的字符又会是什么呢？是否为“1”呢？请看下面的结果：

```
[fgao@ubuntu chapter2]# ./a.out
we write 1 member
We read , ret is 0
```

为什么`fread`什么都没有读取到，返回值是0呢？这是因为上面的代码中，`fwrite`和`fread`操作的是同一个文件指针`fp`，也就是对应的是同一个文件描述符。第一次`fwrite`后，在`tmp.txt`中写入了字符串“123”，同时文件偏移为3，也就是到了文件尾。进行`fread`操作时，既然操作的是同一个文件描述符，自然会共享同一个文件偏移，那么，从文件尾自然读取不到任何数据。

2.6 ferror的返回值

ferror用于告诉用户C库的文件流FILE是否有错误发生。当有错误发生时，**ferror**返回非零值，反之则返回0。那么**ferror**是否会返回不同的错误呢？让我们来看看**ferror**的源码。

```
weak_alias (_IO_ferror, ferror)
int _IO_ferror (fp)
    _IO_FILE* fp;
{
    int result;
    /* 检查文件流的有效性，失败则返回

EOF */
    CHECK_FILE (fp, EOF);
    _IO_flockfile (fp);
    result = _IO_ferror_unlocked (fp);
    _IO_funlockfile (fp);
    return result;
}
```

进入 **_IO_ferror_unlocked**，代码如下：

```
#define _IO_ferror_unlocked(__fp) (((__fp)->_flags & _IO_ERR_SEEN) != 0)
#define _IO_ERR_SEEN 0x20
```

从源码上可以看出**ferror**有两个返回值：

- 当文件流FILE*fp非法时，返回EOF（-1）。
- 当文件流FILE*fp前面的操作发生错误时，返回1。

并且由于文件流的错误只是使用一个标志位 **_IO_ERR_SEEN** 来表示的，因此**ferror**的返回值就不可能针对不同的错误返回不同的值了。

2.7 clearerr的用途

2.6节中的ferror用于检测文件流是否有错误发生，而clearerr用于清除文件流的文件结束位和错误位。

查看clearerr的实现，代码如下：

```
#define clearerr_unlocked(x) clearerr (x)
void
clearerr_unlocked (fp)
    FILE *fp;
{
    CHECK_FILE (fp, /*nothing*/);
    _IO_clearerr (fp);
}
#define _IO_clearerr(FP) ((FP)->_flags &= ~(_IO_ERR_SEEN|_IO_EOF_SEEN))
```

可见，clearerr可以清除文件流中的文件结尾标志和错误标志。

但是清除错误标志又有什么用处呢？按照某些资料上的描述，当文件流读到文件尾时，文件流会被设置上EOF标志。如果不使用clearerr清除EOF标志，即使有新的数据，也无法读取成功。

让我们写个程序来验证一下：

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    FILE *fp = fopen("./tmp.txt", "r");
    if (!fp) {
        printf("Fail to fopen\n");
        return -1;
    }
    while (1) {
        int c = getc(fp);
        if (feof(fp)) {
            printf("reach feof\n");
        }
    }
    return 0;
}
```

为了满足前面所说的测试情况，我们使用gdb来控制程序，代码如下：

31	int c = getc(fp);
(gdb)	
33	if (feof(fp)) {
(gdb) n	
34	printf("reach feof\n");

现在，文件流fp已经读到了文件尾，被设置上了EOF标志。接下来向tmp.txt追加一个字母‘a’。

```
[fgao@fgao chapter3]#echo "a" >> tmp.txt
```

继续gdb，getc仍然可以继续读取，并获得新数据。

```
(gdb) n
31             int c = getc(fp);
(gdb)
33             if (feof(fp)) {
(gdb) p c
$1 = 97
(gdb) p /c c
$2 = 97 'a'
```

继续下一步：

```
(gdb) n
34             printf("reach feof\n");
```

我们可以发现虽然此时文件流`fp`仍然是被设置了EOF标志，但是依然能够成功读取数据。这与某些资料的描述不符，这就应对了那句老话“尽信书不如无书”，对于一些资料的结论，不要完全相信，而是要通过自己的实践来验证。

下面回到glibc的源码，查看`_IO_getc`，从代码中了解为什么是这样的结果。

```
int
_IO_getc (fp)
    FILE *fp;
{
    int result;
    /* 检查

fp */
    CHECK_FILE (fp, EOF);
    _IO_acquire_lock (fp);
    result = _IO_getc_unlocked (fp);
    _IO_release_lock (fp);
    return result;
}
/*只有定义了
```

`IO_DEBUG`,

`CHECK_FILE`才会检查

`_IO_file_flags`标志,

当其不为

0时，则返回错误值。对于

`fgetc`即为

```
EOF
*/
#ifdef IO_DEBUG
#define CHECK_FILE(FILE, RET) \
    if ((FILE) == NULL) { MAYBE_SET_EINVAL; return RET; } \
```

```
    else { COERCE_FILE(FILE); \
          if (((FILE)->_IO_file_flags & _IO_MAGIC_MASK) != _IO_MAGIC) \
            { MAYBE_SET_EINVAL; return RET; }}
#else
# define CHECK_FILE(FILE, RET) COERCE_FILE (FILE)
#endif
```

从glibc的源码中可以发现，文件流FILE的错误标志位只有在打开IO_DEBUG的情况下才会对后面的I/O调用产生影响：在有错误标志位的时候，后面的I/O调用都会直接返回EOF。而一般情况下，IO_DEBUG这个宏是没有定义的。

2.8 小心fgetc和getc

fgetc和getc是两个定义得很不友好的函数，其函数名中的getc很容易让使用者误以为其返回值是char字符。实际上两个函数的接口定义如下：

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE *stream);
```

两者的返回值都是int类型。为什么要用int类型作为返回值呢？因为当文件流读到文件尾时，需要返回EOF值。C99标准中规定了EOF为一个int类型的负数常量，并没有规定具体的值。在glibc中，EOF被定义为-1且char为有符号数。但是不能排除某些实现将EOF定义为其负值，甚至可能因为不遵守C99标准，EOF的值有可能超过char的表示范围。因此，为了代码的健壮性和可移植性，在使用fgetc和getc时，应使用int类型的变量保存其返回值。

2.9 注意fread和fwrite的返回值

fread和fwrite的声明代码如下：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

这两个函数原型很容易让人产生误解。当看到返回值类型为size_t时，人们很有可能理解为fread和fwrite会返回成功读取或写入的字节数，然而实际上其返回的是成功读取或写入的个数，即有多少个size大小的对象被成功读取或写入了。而参数nmemb则用于指示fread或fwrite要执行的对象个数。

看看下面的示例代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    const char str[] = "123456789";
    FILE *fp = fopen("tmp.txt", "w");
    size_t size = fwrite(str, strlen(str), 1, fp);
    printf("size is %d\n", size);
    fclose(fp);
    return 0;
}
```

这段代码的输出为：

```
size is 1
```

结果并不是写入的字符串长度9，而是返回写入的对象个数1。其原因是参数ptr指示的是要写入对象的地址，size为每个对象的字节数，nmemb为有多少个要写入的对象。

将上面的代码稍微变换一下，将fwrite的语句改为：

```
size_t size = fwrite(str, 1, strlen(str), fp);
```

这时程序的输出就变为：

```
size is 9
```

其原因在于，参数size表示每个对象的字节数是1字节，nmemb表示要写入9个对象，因此返回值就变为9了。

2.10 创建临时文件

在项目中经常会需要生成临时文件，用于保存临时数据，创建管道文件、Unix域socket等。为了不与已有的文件同名，或者避免与其他临时文件相冲突，有些朋友可能会选择利用进程id、时间戳等来生成临时文件名。其实，C库已经提供了生成临时文件的接口。下面对生成临时文件的各种方法进行分析对比。先来看看tmpnam方式，代码如下：

```
#include <stdio.h>
char *tmpnam(char *s);
```

tmpnam会返回一个目前系统不存在的临时文件名。当s为NULL时，返回的文件名保存在一个静态的缓存中，因此再次调用tmpnam时，新生成的文件名会覆盖上一次的结果。当s不为NULL时，生成的临时文件名会保存在s中，因此要求s至少要有C库规定的L_tmpnam大小。C库同时还规定tmpnam产生的临时文件的路径以P_tmpdir开头——glibc中P_tmpdir定义为/tmp。

从上面的描述中可以清楚地发现tmpnam的缺点：

- 当s为NULL时，tmpnam不是线程安全的。
- tmpnam生成的临时文件名，必须位于固定的路径下（/tmp）。
- 使用tmpnam创建临时文件不是一个原子行为，需要先生成临时文件名，然后调用其他I/O函数创建文件。这有可能会在创建文件时，该文件已经存在。

再来看看tmpfile方式：

```
#include <stdio.h>
FILE *tmpfile(void);
```

tmpfile返回一个以读写模式打开的、唯一的临时文件流指针。当文件指针关闭或程序正常结束时，该临时文件会被自动删除。

tmpfile直接返回临时的文件流指针——这个自然避免了tmpnam中潜在的线程安全问题，同时还避免了将生成文件名和创建文件分为两个步骤来执行的行为。那么tmpfile是否真的实现了原子地创建临时文件？让我们看一下tmpfile的实现，代码如下：

```
FILE *
tmpfile (void)
{
    char buf[FILENAME_MAX];
    int fd;
    FILE *f;
    if (!_path_search (buf, FILENAME_MAX, NULL, "tmpf", 0))
        return NULL;
```

```
int flags = 0;
#ifdef FLAGS
    flags = FLAGS;
#endif
fd = __gen_tempname (buf, 0, flags, __GT_FILE);
if (fd < 0)
    return NULL;
/* Note that this relies on the UNIX semantics that
   a file is not really removed until it is closed.  */
(void) __unlink (buf);
if ((f = __fdopen (fd, "w+b")) == NULL)
    __close (fd);
return f;
}
```

乍一看，`tmpfile`是通过`__path_search`先产生临时文件名，然后再创建该文件，最后通过文件句柄生成文件流指针。这样的过程看上去好像并不是原子的。下面，让我们深入到`__gen_tempname`中一探究竟。

```
case __GT_FILE:
    fd = __open (tmpl,
                (flags & ~O_ACCMODE)
                | O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    break;
```

在创建临时文件时，C库使用了`open`函数的`O_CREAT`和`O_EXCL`标志组合，这点保证了文件的原子性创建，从而使`tmpfile`创建临时文件的行为是原子的。但`tmpfile`也有一个缺点，与`tmpnam`相同，这个临时文件只能生成在固定的路径下（`/tmp`），并且其有可能因为文件名称冲突而失败返回`NULL`。

那么，有没有可以给临时文件指定目录的方法呢？下面请看`mkstemp`，代码如下：

```
#include <stdlib.h>
int mkstemp(char *template);
```

`mkstemp`会根据`template`创建并打开一个独一无二的临时文件。`template`的最后6个字符必须是“XXXXXX”。`glibc`库会生成一个独一无二的后缀来替换“XXXXXX”，因此要求`template`必须是可以修改的。

`mkstemp`执行成功后会返回创建的临时文件的文件描述符，失败时则返回-1。下面看一下`mkstemp`的实现。

```
int #mkstemp (template)
    char *template;
{
    return __gen_tempname (template, 0, 0, __GT_FILE);
}
```

进入`__gen_tempname`后：

```
int #__gen_tempname (char *tmpl, int suffixlen, int flags, int kind)
{
    int len;
    char *XXXXXX;
    static uint64_t value;
    uint64_t random_time_bits;
    unsigned int count;
```

```

    int fd = -1;
    int save_errno = errno;
    struct stat64 st;
#define ATTEMPTS_MIN (62 * 62 * 62)
    /* The number of times to attempt to generate a temporary file. To
       conform to POSIX, this must be no smaller than TMP_MAX. */
    #if ATTEMPTS_MIN < TMP_MAX
        unsigned int attempts = TMP_MAX;
    #else
        unsigned int attempts = ATTEMPTS_MIN;
    #endif
    /* 检查

```

template的合法性，检查长度及结尾的

XXXXXX字符

```

    */
    len = strlen (tmpl);
    if (len < 6 + suffixlen || memcmp (&tmpl[len - 6 - suffixlen], "XXXXXX", 6))
    {
        __set_errno (EINVAL);
        return -1;
    }
    /* 得到结尾

```

XXXXXX起始位置

```

    */
    XXXXXX = &tmpl[len - 6 - suffixlen];
    /* 得到“随机”数据

```

```

    */
#ifdef RANDOM_BITS
    RANDOM_BITS (random_time_bits);
#else
    #if HAVE_GETTIMEOFDAY || _LIBC
    {
        struct timeval tv;
        __gettimeofday (&tv, NULL);
        random_time_bits = ((uint64_t) tv.tv_usec << 16) ^
            tv.tv_sec;
    }
    #else
        random_time_bits = time (NULL);
    #endif
    #endif
    /* 根据上面的伪随机数和进程

```

pid生成

```

value */
value += random_time_bits ^ __getpid ();
    /*
    根据

```

value得到唯一的临时文件名，如有重复则加上

7777继续。

最多重复

attempts次。

```
*/
for (count = 0; count < attempts; value += 7777, ++count)
{
    uint64_t v = value;
    /*
    letters是
```

26个英文大小写加上

10个阿拉伯数字，为

62个大小的字符数组。因此使用

62作为除数，

以得到随机字符。

```
*/
XXXXXX[0] = letters[v % 62];
v /= 62;
XXXXXX[1] = letters[v % 62];
v /= 62;
XXXXXX[2] = letters[v % 62];
v /= 62;
XXXXXX[3] = letters[v % 62];
v /= 62;
XXXXXX[4] = letters[v % 62];
v /= 62;
XXXXXX[5] = letters[v % 62];
switch (kind)
{
    case __GT_FILE:
        /* 这是
```

mkstemp的情况，利用

O_CREAT|O_EXCL创建唯一文件

```
*/
        fd = __open (tmpl,
            (flags & ~O_ACCMODE)
            | O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
        break;
    }
    if (fd >= 0)
    {
        /* 成功创建了文件，恢复原来的
```

errno，并返回创建的文件描述符

```

fd */
    __set_errno (save_errno);
    return fd;
}
else if (errno != EEXIST) {
/* 如失败的原因不是因为文件已经存在的时候，则直接返回。

*/
    return -1;
}
/* 如果是其他原因，则会重新生成新的文件名，并再次尝试重建

*/
}
/* 将

errno设置为

EEXIST，即文件已经存在

*/
__set_errno (EEXIST);
return -1;
}

```

综上所述，在需要使用临时文件时，不推荐使用`tmpnam`，而要用`tmpfile`和`mkstemp`。前者的局限在于不能指定路径，并且在文件名称冲突时会返回失败。后者可以由调用者来指定路径，并且在文件名称冲突时，会自动重新生成并重试。

除了上面介绍的几种方法，Linux环境还提供了这些接口的一些变种：`tempnam`、`mkostemp`、`mkstemp`等，分别对其原始形态进行了扩展，详细区别可以直接查看Linux手册。

第3章 进程环境

进程是操作系统运行程序的一个实例，也是操作系统分配资源的单位。在Linux环境中，每个进程都有独立的进程空间，以便对不同的进程进行隔离，使之不会互相影响。深入理解Linux下的进程环境，可以帮助我们写出更健壮的代码。

3.1 main是C程序的开始吗

在编写C程序的时候，都是从main函数开始，然而main函数真的是C程序的入口吗？让我们来看看下面的程序：

```
#include <stdlib.h>
#include <stdio.h>
static void __attribute__((constructor)) before_main(void)
{
    printf("Before main...\n");
}
int main(void)
{
    printf("Main!\n");
    return 0;
}
```

其执行结果为：

```
Before main...
Main!
```

从运行结果中，可以发现before_main是在进入main函数之前被调用的，这点对于C语言的初学者来说似乎有点难以接受。究竟是谁调用的before_main呢？怎么还没有进入main就可以有代码被执行呢？

回忆一下第0章所讲的基础知识，在编译的过程中可以使用-v来详细地显示编译的过程。在此，截取gcc 4_1_main_stack.c-v输出的一部分结果，如下所示：

```
/usr/libexec/gcc/i686-redhat-linux/4.6.3/collect2 --build-id --no-add-needed
--eh-frame-hdr -m elf_i386 --hash-style=gnu -dynamic-linker /lib/ld-linux.
so.2 /usr/lib/gcc/i686-redhat-linux/4.6.3/../../../../crt1.o /usr/lib/gcc/i686-
redhat-linux/4.6.3/../../../../crti.o /usr/lib/gcc/i686-redhat-linux/4.6.3/
crtbegin.o -L/usr/lib/gcc/i686-redhat-linux/4.6.3 -L/usr/lib/gcc/i686-
redhat-linux/4.6.3/../../../../tmp/cc3tzF7V.o -lgcc --as-needed -lgcc s --no-
as-needed -lc -lgcc --as-needed -lgcc s --no-as-needed /usr/lib/gcc/i686-
redhat-linux/4.6.3/crtend.o /usr/lib/gcc/i686-redhat-linux/4.6.3/../../../../
crti.o
```

可以看到，在链接生成最后的可执行文件时，有大量的C库二进制文件参与进来，如crt1.o、crti.o等。可见最终的可执行文件，除了我们编写的这个简单的C代码以外，还有大量的C库文件参与了链接，并包含在最终的可执行文件中。这个“组装”的过程，是由链接器ld的链接脚本来决定的。在没有指定链接脚本的情况下，会使用ld的默认脚本，可以通过ld-verbose来查看，下面截取了对我们有用的部分输出：

```
/* Script for -z combrelloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
```

这里定义了输出的文件格式、目标机器的类型，以及重要的信息和程序的入口ENTRY（_start）。

```

.ctors      :
{
    /* gcc uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.  Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.  The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*crtbegin?.o(.ctors))
    /* We don't want to include the .ctor section from
       the crtend.o file until after the sorted ctors.
       The .ctor section from the crtend file contains the
       end of ctors marker and it must be last */
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
}

```

这里定义了.ctors section，而我们的例子中before_main函数使用的gcc扩展属性

`__attribute__((constructor))`就是将函数对应的指令归属于.ctors section中。

下面我们来追溯一下Linux可执行程序完整的启动过程。前面的链接脚本明确了入口为_start。在32位的x86平台中，_start位于sysdeps/i386/start.S中。

```

.text
.globl _start
.type _start,@function
_start:
    /* Clear the frame pointer.  The ABI suggests this be done, to mark
       the outermost frame obviously.  */
    xorl %ebp, %ebp
    /* Extract the arguments as encoded on the stack and set up
       the arguments for `main': argc, argv.  envp will be determined
       later in __libc_start_main.  */
    popl %esi /* Pop the argument count.  */
    movl %esp, %ecx /* argv starts just at the current stack top. */
    /* Before pushing the arguments align the stack to a 16-byte
       (SSE needs 16-byte alignment) boundary to avoid penalties from
       misaligned accesses.  Thanks to Edward Seidl <seidl@janed.com>
       for pointing this out.  */
    andl $0xffffffff, %esp
    pushl %eax /* Push garbage because we allocate
                28 more bytes.  */
    /* Provide the highest stack address to the user code (for stacks
       which grow downwards).  */
    pushl %esp
    pushl %edx /* Push address of the shared library
                termination function.  */
    /* Push address of our own entry points to .fini and .init.  */
    pushl $__libc_csu_fini
    pushl $__libc_csu_init
    pushl %ecx /* Push second argument: argv.  */
    pushl %esi /* Push first argument: argc.  */
    pushl $BP_SYM (main)
    /* Call the user's main function, and exit with its value.
       But let the libc call main.  */
    call BP_SYM (__libc_start_main)

```

上面列出的虽然是汇编代码，但是每一行都有清楚的注释，这段代码主要是为程序的运行创建好运行环境，其中需要注意的是，__libc_csu_fini和__libc_csu_init都被作为参数传给了__libc_start_main。从这两个函数的名字上可以推测它们是用来处理退出和初始化阶段的函数，那么.ctors section中的函数很可能就是由__libc_csu_init来调用的。

我们先来关注__libc_csu_init是在何时被调用的，然后再分析其实现。上面的汇编代码将这两个函数作为参数传递给了__libc_start_main，然后又调用了generic_start_main函数。这个函数初始化了C库所

需要的环境，如环境变量、函数栈、多线程环境等，最后调用main函数——进入普通应用程序的真正入口。而在此之前，以下代码先被执行：

```
/* Register the destructor of the program, if any. */
if (fini)
    __cxa_atexit ((void (*)(void *)) fini, NULL, NULL);
if (!__init)
    (*__init) (argc, argv, __environ MAIN_AUXVEC_PARAM);
```

init即为__libc_csu_init，上面的代码保证了__libc_csu_init在main之前被调用。那么.ctors的函数又是如何被__libc_csu_init调用的呢？篇幅所限，在此我们就不罗列代码，只给出其调用流程：

__libc_csu_init->_init->__libc_global_ctors。

```
void
__libc_global_ctors (void)
{ /* Call constructor functions. */
  run_hooks (__CTOR_LIST__);
}
static inline void
run_hooks (void (*const list[]) (void))
{
  while ((*++list)
    (**list) ());
}
static void (*const __CTOR_LIST__[1]) (void)
  __attribute__((used, section(".ctors")))
  = { (void (*)(void)) -1 };
```

__CTOR_LIST__是一个函数指针数组，数组的大小为1。该数组使用gcc的扩展属性，使__CTOR_LIST__位于.ctors section中。因此，在上面的代码中，__libc_global_ctors将__CTOR_LIST__传递给了run_hooks，实际上就是将.ctors section的起始地址传递给了run_hooks。而__CTOR_LIST__位于.ctors的第一个位置，其本身并不是一个真正的.ctors属性函数，因此run_hooks的while (*++list)先执行自增操作，即跳过了__CTOR_LIST__。

可以通过反汇编查看二进制的可执行程序来验证：

```
080483e4 <before_main>:
80483e4: 55                push    %ebp
80483e5: 89 e5             mov     %esp,%ebp
80483e7: 83 ec 18          sub     $0x18,%esp
80483ea: c7 04 24 e0 84 04 movl    $0x80484e0,(%esp)
80483f1: e8 22 ff ff ff    call    8048318 <puts@plt>
80483f6: c9               leave   %ebp
80483f7: c3               ret
```

可以看到，函数before_main的地址为0x080483e4。然后使用objdump来查看.ctors section：

```
objdump -s -j .ctors a.out
a.out:          file format elf32-i386
Contents of section .ctors:
8049f08 ffffffff e4830408 00000000      ....
```

可以看到，`.ctors` section的第一个元素即上文中的`__CTOR_LIST__`，第二个元素为`before_main`——由于x86是小端CPU，因此`0xe4830408`实际上表示的地址值为`0x080483e4`。

需要注意的是，在新版本的gcc中，`.ctors`属性的函数并不会位于`.ctors` section中，而是被gcc合并到了`.init_array` section中。下面来看一下这种情况下的objdump输出：

```
[fgao@fgao chapter3]#objdump -s -j .ctors a.out
a.out:          file format elf32-i386
Contents of section .ctors:
8049600 ffffffff 00000000          .....
```

可以看到，在`.ctors` section中，没有任何有效的`.ctors`函数，然后我们来看看`.init_array` section：

```
[fgao@fgao chapter3]#objdump -s -j .init_array a.out
a.out:          file format elf32-i386
Contents of section .init_array:
80495fc b4830408          .....
```

保存在`.init_array` section中的函数调用机制与之前分析的`.ctors` section机制类似，在此就不再重复了。感兴趣的朋友可以自行分析。



说明 与`constructor`属性对应的，还有`desconstructor`属性。拥有`desconstructor`属性的函数，会在`main`结束之后被调用。

3.2 “活雷锋”exit

在刚刚学习C语言的时候，我们被告知分配内存以后，如果不使用free来释放内存，就会造成内存的泄漏。同样，打开文件以后，如果忘记close也会造成资源的泄漏。那么，在进程退出以后，这些资源是否真的泄漏了呢？

当进程正常退出时，会调用C库的exit；而当进程崩溃或被kill掉时，C库的exit则不会被调用，只会执行内核退出进程的操作。

首先，我们来分析C库的退出函数exit，代码如下：

```
void
exit (int status)
{
    __run_exit_handlers (status, &__exit_funcs, true);
}
```

C库的exit主要用来执行所有注册的退出函数，比如使用atexit或on_exit注册的函数。执行完注册的退出函数后，__run_exit_handlers会调用_exit，代码如下：

```
void
_exit (status)
int status;
{
    while (1)
    {
#ifdef NR_exit_group
        INLINE_SYSCALL (exit_group, 1, status);
#endif
        INLINE_SYSCALL (exit, 1, status);
#ifdef ABORT_INSTRUCTION
        ABORT_INSTRUCTION;
#endif
    }
}
```

上面的代码很简单，当平台有exit_group时，就调用exit_group，否则就调用exit。从Linux内核2.5.35版本以后，为了支持线程，就有了exit_group。这个系统调用不仅仅是用于退出当前线程，还会让所有线程组的线程全部退出。

下面来看看系统调用exit_group的实现：

```
SYSCALL_DEFINE1(exit_group, int, error_code)
{
    /* do_group_exit做真正的工作

    ...

    */
    do_group_exit((error_code & 0xff) << 8);
    /* NOTREACHED */
    return 0;
}
NORET_TYPE void
do_group_exit(int exit_code)
{
    struct signal_struct *sig = current->signal;
    BUG_ON(exit_code & 0x80); /* core dumps don't get here */
    /* 检查该线程组是否正在退出，如果条件为真，则不需要设置线程组退出的条件，直接执行本线程

    task退出流程

    do_exit即可

    */
    if (signal_group_exit(sig))
        exit_code = sig->group_exit_code;
    else if (!thread_group_empty(current)) { /* 线程组不为空

    ...

    */struct sighand_struct *const sighand = current->sighand;
        spin_lock_irq(&sighand->siglock);
        /* 标准的双重条件检查机制，因为第一次检查

    signal_group_exit时为假，但是另外一个线程

    ...

    已经拿到锁，并设置了状态。当拿到锁的时候，需要再次检查

    */
    if (signal_group_exit(sig)) {
        /* Another thread got here before we took the lock. */
        exit_code = sig->group_exit_code;
    }
    else {
        /* 设置线程组的退出值和退出状态

    ...

    */
        sig->group_exit_code = exit_code;
        sig->flags = SIGNAL_GROUP_EXIT;
        /* 使用

    SIGKILL干掉“线程组的其他线程
```

```

*/
        zap_other_threads(current);
    }
    spin_unlock_irq(&sighand->siglock);
}
/* 真正的退出动作，退出当前线程

task */
do_exit(exit_code);
/* NOTREACHED */
}

```

下面来看看do_exit的实现：

```

NORET_TYPE void do_exit(long code)
{
    struct task_struct *tsk = current;
    int group_dead;
    profile_task_exit(tsk);
    WARN_ON(blk_needs_flush_plug(tsk));
    /* 中断上下文不能使用退出，因为没有进程上下文

```

```

*/
    if (unlikely(in_interrupt()))
        panic("Aieee, killing interrupt handler!");
    /* pid%

```

0. 即内核的

idle进程。这个

task也是不应该退出的

```

*/
    if (unlikely(!tsk->pid))
        panic("Attempted to kill the idle task!");
    /*
     * If do_exit is called because this processes oopsed, it's possible
     * that get_fs() was left as KERNEL_DS, so reset it to USER_DS before
     * continuing. Amongst other possible reasons, this is to prevent
     * mm_release()->clear_child_tid() from writing to a user-controlled
     * kernel address.
     */
    set_fs(USER_DS);
    /* 如果

```

task正在被跟踪如

gdb，则发送

ptrace事件

```

*/
    ptrace_event(PTRACE_EVENT_EXIT, code);
    validate_creds_for_do_exit(tsk);
    /*
     * We're taking recursive faults here in do_exit. Safest is to just
     * leave this task alone and wait for reboot.
     */
    /* 当

```

task退出的时候，会被设置上

PF_EXITING标志。如果发现此时

flags已经设置了该标志，则说

明发生了错误。此时就要按照注释所说的，最安全的方法是什么都不做，通知并等待重启

```

*/
    if (unlikely(tsk->flags & PF_EXITING)) {
        printk(KERN_ALERT
            "Fixing recursive fault but reboot is needed!\n");
        /*
         * We can do this unlocked here. The futex code uses
         * this flag just to verify whether the pi state
         * cleanup has been done or not. In the worst case it
         * loops once more. We pretend that the cleanup was
         * done as there is no way to return. Either the
         * OWNER_DIED bit is set by now or we push the blocked
         * task into the wait for ever nirwana as well.
         */
        tsk->flags |= PF_EXITPIDONE;
        /* 将当前

```

task设置为不可中断的状态，然后放弃

CPU.

```

*/
    set_current_state(TASK_UNINTERRUPTIBLE);
    schedule();
}
/*如果当前

```

task是中断线程，即每个

CPU中断由一个线程来处理，则设置对应的中断停止来唤醒本线程。这

是一个编译选项，默认情况下是关闭的。

```

*/
exit_irq_thread();
/* 给

```

task设置退出标志

```

PF_EXITING */
__exit_signals(tsk); /* sets PF_EXITING */
/*
 * tsk->flags are checked in the futex code to protect against
 * an exiting task cleaning up the robust pi futexes.
 */
smp_mb();
raw_spin_unlock_wait(&tsk->pi_lock);
if (unlikely(in_atomic()))
    printk(KERN_INFO "note: %s[%d] exited with preempt_count %d\n",
           current->comm, task_pid_nr(current),
           preempt_count());
acct_update_integrals(tsk);
/* sync mm's RSS info before statistics gathering */
/* 该

```

task有自己的内存空间

```

*/
if (tsk->mm)
    sync_mm_rss(tsk, tsk->mm); //更新内存统计计数

```

/* 判断整个线程组是否都已经退出。

```

*/
group_dead = atomic_dec_and_test(&tsk->signal->live);
if (group_dead) {
    /* 取消高精度定时器

```

```

*/
hrtimer_cancel(&tsk->signal->real_timer);
/* 删除

```

task的内部定时器，对应系统调用

getitimer和

```

setitimer */
exit_itimers(tsk->signal);
if (tsk->mm)
    setmax_mm_hiwater_rss(&tsk->signal->maxrss, tsk->mm);
}
acct_collect(code, group_dead);
/* 如果整个线程组都已经退出，则释放授权资源

```

```

*/
if (group_dead)
    tty_audit_exit();
if (unlikely(tsk->audit_context))
    audit_free(tsk);
/* 设置

```

task的退出值

```

*/
tsk->exit_code = code;
/* 释放任务统计资源

```

```

*/
taskstats_exit(tsk, group_dead);
/*
释放

```

task的内存空间。

task使用的所有内存页都由内核来维护。对于用户程序，如果忘记释放申请的内存。

则只会造成用户程序无法再使用该内存，因为内核认为该内存仍然在被用户程序使用。当

task退出时，内核

会负责释放所有的内存地址。因此当进程退出时，所有申请的内存都会被释放，不会有任何的内存泄漏。

```
*/
exit_mm(tsk);
if (group_dead)
    acct_process();
trace_sched_process_exit(tsk);
/*
检查是否释放了
```

semaphore资源，如没有释放则执行

semaphore的

undo操作。这点用于保证在进程意外退

出时，能恢复

semaphore的正确状态，也可以用于预防错误的程序逻辑所导致的

semaphore释放操作遗漏。

```
*/
exit_sem(tsk);
/* 释放共享内存

*/
exit_shm(tsk);
/*
如果文件资源没有被共享，则释放所有的文件资源。即使用户程序有文件泄漏也不必担心，一旦
```

task退

出，文件资源都会得到正确的释放~因为内核维护了所有的、打开的文件。

```
*/
exit_files(tsk);
/* 释放
```

task的文件系统资源，如当前目录、根目录等

```
*/
exit_fs(tsk);
check_stack_usage();
/* 释放
```

task资源，如

TSS段等

```
*/
exit_thread();
/*
* Flush inherited counters to the parent ~ before the parent
* gets woken up by child-exit notifications.
*
* because of cgroup mode, must be called before cgroup_exit()
*/
perf_event_exit_task(tsk);
/* 从控制组退出，并释放相关资源
```

```
*/
cgroup_exit(tsk, 1);
/* 如果线程组都已经退出，则断开控制终端即
```

```
tty */
if (group_dead)
    disassociate_ctty(1);
/* 后面仍然是一些
```

task退出的清理工作，因与本节关系不大，所以在此不再一一列出了

```
*/
...

}
```

从exit的源码可以得知，即使应用程序在应用层有内存泄漏或文件句柄泄漏也不必担心，当进程退出时，内核的exit_group调用将会默默地在后面做着清理工作，释放所有内存，关闭所有文件，以及其

3.3 atexit介绍

3.3.1 使用atexit

atexit用于注册进程正常退出时的回调函数。若注册了多个回调函数，最后的调用顺序与注册顺序相反，与我们熟悉的栈操作类似，先入后出。

```
#include <stdlib.h>
int atexit(void (*function) (void));
```

下面来看一个简单的例子：

```
#include <stdlib.h>
#include <stdio.h>
static void callback1(void)
{
    printf("callback1\n");
}
static void callback2(void)
{
    printf("callback2\n");
}
static void callback3(void)
{
    printf("callback3\n");
}
int main(void)
{
    atexit(callback1);
    atexit(callback2);
    atexit(callback3);
    printf("main exit\n");
    return 0;
}
```

它的运行结果如下：

```
main exit
callback3
callback2
callback1
```

从上面的代码输出可以看出，我们顺序地注册callback1、callback2和callback3，当进程退出时，其调用顺序为callback3、callback2和callback1。

3.3.2 atexit的局限性

3.3.1节介绍atexit的基本用法时提到过，使用atexit注册的退出函数是在进程正常退出时，才会被调用。这里的正常退出是指，使用exit退出或使用main中最后的return语句退出。若是因为收到信号而导致程序退出，atexit注册的退出函数则不会被调用。下面我们通过一个测试程序来验证这一观点：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
static void callback1(void)
{
    printf("callback1\n");
}
int main(void)
{
    atexit(callback1);
    while (1) {
        sleep(1);
    }
    printf("main exit\n");
    return 0;
}
```

然后编译运行，使用另一个控制台给其发送信号：

```
[fgao@fgao ik8]#killall atexit_signal
```

我们会发现atexit注册的退出函数并没有被调用：

```
[fgao@fgao chapter3]#./atexit_signal;
Terminated
```

为什么只有在正常退出的时候，atexit注册的退出函数才能被调用呢？下面我们来分析atexit的源码实现，就可以得到答案了。

3.3.3 atexit的实现机制

让我们带着疑问来分析glibc中的atexit源码：

```
int
#ifdef atexit
attribute_hidden
#endif
atexit (void (*func) (void))
{
    /* __dso_handle是动态共享对象的句柄，此处可以略过

    */
    return __cxa_atexit ((void (*) (void *)) func, NULL,
        &__dso_handle == NULL ? NULL : __dso_handle);
}
int
__cxa_atexit (void (*func) (void *), void *arg, void *d)
{
    /* __exit_funcs为退出函数的链表

    */ return __internal_atexit (func, arg, d, &__exit_funcs);
}
int
attribute_hidden
__internal_atexit (void (*func) (void *), void *arg, void *d,
    struct exit_function_list **listp)
{
    /* 在退出函数链表中，得到一个新的节点

    */
    struct exit_function *new = __new_exitfn (listp);
    if (new == NULL)
        return -1;
#ifdef PTR_MANGLE
    PTR_MANGLE (func);
#endif
    /* 初始化这个节点，将函数及其参数赋给这个节点

    */
    new->func.cxa.fn = (void (*) (void *, int)) func;
    new->func.cxa.arg = arg;
    new->func.cxa.dso_handle = d;
    atomic_write_barrier ();
    new->flavor = ef_cxa;
    return 0;
}
```

上面的代码揭示了atexit是如何把函数注册到退出函数链表中的。那么，这些函数又是何时被调用的呢？回忆atexit的介绍，退出注册函数只有在程序正常退出或调用exit时才会被执行。程序正常退出时，系统就会调用exit。因此，问题的关键就在于exit函数了：

```
void
exit (int status)
{
    __run_exit_handlers (status, &__exit_funcs, true);
}
```

在这里，__run_exit_handlers会遍历__exit_funcs，一一调用注册的退出函数，在此就不再罗列其代码了。从atexit的实现机制上进行分析，我们可以得出atexit的实现是依赖于C库的代码的。当进程收到

信号时，如果没有注册对应的信号处理函数，那么内核就会执行信号的默认动作，一般是直接终止进程。这时，进程的退出完全由内核来完成，自然不会调用到C库的**exit**函数，也就无法调用注册的退出函数了。

3.4 小心使用环境变量

Linux环境下，程序在启动的时候都会从shell环境下继承当前的环境变量，如PATH、HOME、TZ等。我们也可以通过C库的接口来增加、修改或删除当前进程的环境变量，示例如下：

```
#include <stdlib.h>
int putenv(char *string);
```

putenv用于增加或修改当前的环境变量。string的格式为“名字=值”。如果当前环境变量没有该名称的环境变量，则增加这个新的环境变量；如果已经存在，则使用新值。看似功能很简单，但实际上使用这个接口时，却很容易犯错。请看下面的代码：

```
#include <stdlib.h>
#include <stdio.h>
static void set_env_string(void)
{
    char test_env[] = "test_env=test";
    if (0 != putenv(test_env)) {
        printf("fail to putenv\n");
    }
    printf("1. The test_env string is %s\n", getenv("test_env"));
}
static void show_env_string(void)
{
    printf("2. The test_env string is %s\n", getenv("test_env"));
}
int main()
{
    set_env_string();
    show_env_string();return 0;
}
```

然后编译，查看输出结果：

```
1. The test_env string is test
2. The test_env string is (null)
```

结果有点出人意料，为什么在set_env_string中可以得到我们设置的环境变量，而在show_env_string中却不行呢？

原因在于使用putenv添加环境变量时，参数直接被当作环境变量的一部分了。对于本例而言，set_env_string中的test_env数组直接被环境变量引用了。而test_env是一个局部变量，在执行set_env_string的时候，test_env已经不存在了，对应栈上的内存会在后面的函数调用中使用，并存入其他值。因此，在进入show_env_string的时候，就无法得到正确的值了。

笔者曾经修改过一个因为putenv引起的bug，当时也是费了很大一番力气才找到根本原因，所以颇为气愤当时的开发人员为什么在使用putenv的时候，不认真阅读该接口的说明。Martin Golding曾说过一句话“编程的时候，要总是想着那个维护你代码的人会是一个知道你住在哪儿的、有暴力倾向的精神病患者”。

如果非要用`putenv`来设置环境变量，就必须要保证参数是一个长期存在的内容。因此，只能选择全局变量、常量或动态内存等。为了避免犯错，我们应该尽量使用另外一个接口`setenv`，代码如下：

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

参数说明：

·**name**：要加入的环境变量名称。

·**value**：该环境变量的值。

·**overwrite**：用于指示是否覆盖已存在的重名环境变量。

还是使用上文的例子，只不过我们将`putenv`换为`setenv`，代码如下：

```
#include <stdlib.h>
#include <stdio.h>
static void set_env_string(void)
{
    setenv("test_env", "test", 1);
    printf("1. The test_env string is %s\n", getenv("test_env"));
}
static void show_env_string(void)
{
    printf("2. The test_env string is %s\n", getenv("test_env"));
}
int main()
{
    set_env_string();
    show_env_string();
    return 0;
}
```

这次的运行结果就是我们预期的结果了：

```
1. The test_env string is test
2. The test_env string is test
```

3.5 使用动态库

在平时的编程工作中，除了C库，还会用到大量的库文件，其中绝大部分都是以动态库的方式来提供服务的。

3.5.1 动态库与静态库

一般情况下，库文件的开发者会同时提供动态库和静态库两个版本，它们都有各自的优缺点。静态库在链接阶段，会被直接链接进最终的二进制文件中，因此最终生成的二进制文件体积会比较大，但是可以不再依赖于库文件。而动态库并不是被链接到文件中的，只是保存了依赖关系，因此最终生成的二进制文件体积较小，但是在运行阶段需要加载动态库。

3.5.2 编译生成和使用动态库

首先，我们来编译并生成一个动态库：

```
#include <stdlib.h>
#include <stdio.h>
void dynamic_lib_call(void)
{
    printf("dynamic lib call\n");
}
```

编译生成动态库与编译普通的可执行程序略有不同，如下所示：

```
gcc -Wall -shared 4_5_2_dlib.c -o libdlib.so
```

其中多了一个`-shared`选项，该选项用于指示`gcc`生成动态库。

然后再编写一个简单例子，来使用这个动态库，代码如下：

```
#include <stdlib.h>
#include <stdio.h>
extern void dynamic_lib_call(void);
int main(void)
{
    dynamic_lib_call();
    return 0;
}
```

下面我们利用前面的动态库来生成最终的可执行文件`gcc-Wall 4_5_2_main.c-o test_dlib-L./-ldlib`。其中，`-l`用于指示生成文件依赖的库，本例依赖于`libdlib.so`，因此为`-ldlib`；`-L`与`-I`类似，`-L`用于指示`gcc`在哪个目录中查找依赖的库文件。

让我们运行这个`test_dlib`看看结果如何：

```
[fgao@ubuntu chapter3]#./test_dlib
./test_dlib: error while loading shared libraries: libdlib.so: cannot open shared object file: No such file or directory
```

为什么会报告出错，找不到这个`libdlib.so`呢？前面明明已经使用`-L`指定了库文件在当前目录中，并且这个库文件也确实存在于当前目录中啊。这是怎么回事呢？

让我们使用`ldd`来查看`test_lib`的依赖库，代码如下：

```
[fgao@ubuntu chapter3]#ldd test_dlib
linux-gate.so.1 => (0x57785000)
libdlib.so => not found
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75ce000)
/lib/ld-linux.so.2 (0xb7786000)
```

确实显示无法找到`libdlib.so`。原因在于`-L`只是在`gcc`编译的过程中指示库的位置，而在程序运行的时候，动态库的加载路径默认为`/lib`和`/usr/lib`。在Linux环境下，还可以通过`/etc/ld.so.conf`配置文件和环境变量`LD_LIBRARY_PATH`指示额外的动态库路径。

为简单起见，我们在这里将`libdlib.so`复制到`/usr/lib`目录下，再运行`test_dlib`试试：


```
[root@ubuntu lib]#cp /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter3/libdlib.so .
[fgao@ubuntu chapter3]#./test_dlib
dynamic lib call
```

现在./test_dlib顺利执行了，并成功调用了动态库中的dynamic_lib_call函数。

上面的例子中，动态库是由系统自动加载的，所以需要将动态库放在指定的目录下。然而，C库还提供了dlopen等接口来支持手工加载动态库的功能，代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    void *dlib = dlopen("./libdlib.so", RTLD_NOW);
    if (!dlib) {
        printf("dlopen failed\n");
        return -1;
    }
    void (*dfunc) (void) = dlsym(dlib, "dynamic_lib_call");
    if (!dfunc) {
        printf("dlsym failed\n");
        return -1;
    }
    dfunc();
    dlclose(dlib);
    return 0;
}
```

编译代码gcc-Wall 4_5_2_main_mlib.c-ldl-o test_mlib，需要使用-ldl选项来指定依赖的动态库libdl.so。

下面来看一下输出结果：

```
[fgao@ubuntu chapter3]#./test_mlib
dynamic lib call
```

可以看出，我们已经成功地使用手工来加载动态库，并完成了动态库中的函数调用。

介绍完动态库的两种加载方法，我们可以对比一下两者的优缺点。对于自动加载，处理起来比较简单；而手工加载需要编写额外的代码，但正是这些额外的代码提供了更多的动态库的可控性。

3.5.3 程序的“平滑无缝”升级

3.5.1节中，对比了动态库和静态库的优缺点。其中动态库的一个重要优点就是，可执行程序并不包含动态库中的任何指令，而是在运行时加载动态库并完成调用。这就给我们提供了升级动态库的机会。只要保证接口不变，使用新版本的动态库替换原来的动态库，就完成了动态库的升级。更新完库文件以后启动的可执行程序都会使用新的动态库。

这样的更新方法只能够影响更新以后启动的程序，对于正在运行的程序则无法产生效果，因为程序在运行时，旧的动态库文件已经加载到内存中了。我们只能更新位于磁盘上的动态库的物理文件，而不能影响已经位于内存中的库了。

我们是否可以做得更好呢？对于服务程序来说，重启会付出很大的代价并带来糟糕的用户体验。那么，能否让运行中的服务程序也能在升级库以后使用新的指令，做到“平滑无缝”的升级呢？这就需要使用前面介绍的手工加载动态库的方法了。

下面的伪代码将给出一个比较简单的解决方案。

1) 使用一个结构体来管理动态库的接口：

```
struct dlib_manager {void *dlib_handle; //保存动态库的句柄

int (service_func) (void *);int (service_func2) (void *);
} g_dlib_manager;
/* g_dlib_manager作为动态库接口的全局变量

*/
struct dlib_manager *g_dlib_manager;
```

2) 利用dlopen、dlsym等来加载动态库，更新接口。重新申请新的内存，来保存新的动态库接口：

```
/* 更新动态库接口

*/
struct dlib_manager *new_manager = malloc(sizeof(*new_manager));
new_manager->dlib_handle = dlopen("libupgrade.so", RTLD_LAZY);
new_manager->service_func = dlsym(g_dlib_handle, "service_call");
new_manager->service_func2 = dlsym(g_dlib_handle, "service_call2");
/* 在多线程环境下，使用内存屏障，以保证在交换

new_manager和

g_dlib_manager时，
```

`new_manager`已经

完成了赋值

```
*/  
wmb();  
/*交换新指针与当前正在使用的接口指针
```

因为目前，无论是新指针还是旧指针都是有效的接口，所以并不会对业务产生影响

```
*/  
swap(new_manager, g_dlib_manager);  
/*交换完成以后，新的请求都会交由新接口来处理
```

由于当前旧接口仍然可能正在使用中，所以要使用推迟释放或是等待正在服务的接口完成

```
*/  
delay_free(new_manager);
```

3) 在调用服务接口时，要利用局部变量保存服务接口：

```
struct dlib_manager *local_dlib_manager = g_dlib_manager;  
local_dlib_manager->service_func1(data);  
local_dlib_manager->service_func2(data);
```

之所以这里使用局部变量来进行接口调用，是为了避免在调用了一部分接口后，`g_dlib_manager`才发生更新，从而导致前后的服务接口属于不同的动态库，造成不可预料的问题。通过临时变量来保存服务接口，能确保所有接口的一致性。

4) 释放旧接口的关键在于，要保证没有旧接口正在被使用。根据自己的业务，找到一个时间点——在这个时间点上，所有的线程（准确地说是请求流程）都已经服务过一次。这时，新来的请求就会使用新的接口，于是我们也就可以安全地释放旧接口了。

其实整个实现方案是借鉴了Linux内核的RCU实现方式。通过这种方法，可以进行“平滑无缝”的升级，而不影响运行状态下的业务功能。

3.6 避免内存问题

在编程的错误中，内存问题无疑占据了很大的比例。而且内存问题比较难查，出现问题的“案发现场”与真正的“凶手”往往隔着十万八千里，甚至完全没有关系。对于初学者来说，解决这样的问题往往要浪费大量的时间。因此，我们应该在编写代码的初始阶段，就要注意避开某些代码“陷阱”。问题发现得越早，代价也就越小。

3.6.1 尴尬的realloc

对于良好的代码风格，有一项很重要的要求是一个函数只专注于做一件事情。如果该函数像瑞士军刀一样能实现多个功能，那基本上可以断言这不是一个设计良好的函数。

C库中的realloc函数就是一个典型的反面教材：

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

realloc可以将ptr指向的内存调整为size大小。这个功能看上去很明确，其实则不然，其一共有三种不同的行为：

- 参数ptr为NULL，而size不为0，则等同于malloc（size）。
- 参数ptr不为NULL，而size为0，则等同于free（ptr）。
- 参数ptr和size均不为0，其行为类似于free（ptr）； malloc（size）。

有着三种不同行为的realloc，很容易给代码引入bug。下面举一个例子来说明：

```
void * ptr = realloc(ptr, new_size);
if (!ptr) {
    // 错误处理
}

}
```

这里就会因为realloc的第三种行为引入一个bug。当realloc分配内存失败的时候，ptr会返回NULL。但是这时ptr原来指向的内存并没有被释放，而ptr却已经被赋值为NULL了，这就造成了ptr原有内存泄漏。

正确的做法应该是：

```
void * new_ptr = realloc(ptr, new_size);
if (!new_ptr) {
    // 错误处理
}

ptr = new_ptr
```

realloc只有在分配内存成功的情况下，才会让ptr等于new_ptr。这样，在分配内存失败的情况下，ptr指向的内存并不会丢失。

`realloc`使用不当还会引发其他几种bug，在此就不一一罗列了。需要吸取的教训就是，慎用`realloc`，甚至最好不用`realloc`。如果真的需要使用`realloc`，一定要确保在`realloc`的三种行为下代码都可以正常工作。

3.6.2 如何防止内存越界

在日常的编程中，初学者往往会遇到内存越界所引发的问题。其实，通过良好的编程习惯基本上是可以避免内存越界问题的。防范的根本思想在于在对缓冲区（一般为数组）进行拷贝前，要保证复制的长度不要超过缓冲区的空间大小。比如在`memcpy`前，要检查目的地址是否有足够的空间。

使用宏或`sizeof`可保证缓冲长度的一致性；

```
char dst_buf[64];
memcpy(dst_buf, src_buf, 64);
```

当缓冲大小改变为32的时候，需要改动两处代码。一旦忘记修改`memcpy`处的拷贝长度，就会造成内存越界。

对上面的代码进行改善：

```
#define BUF_SIZE    64
char dst_buf[BUF_SIZE];
memcpy(dst_buf, src_buf, BUF_SIZE);
```

或

```
char dst_buf[64];
memcpy(dst_buf, src_buf, sizeof(dst_buf));
```

这样就可以做到缓存大小和复制长度的同步修改。

使用安全的库函数也可以保证复制的长度不超过缓冲区的空间，下面来介绍4种库函数。

1) 使用`strncat`代替`strcat`，代码如下：

```
#include <string.h>
char *strncat(char *dest, const char *src, size_t n);
```

从`src`中最多追加`n`个字符到`dest`字符串的后面。需要注意的是，当`src`包含`n`个以上的字符时，`dest`的空间至少为`strlen(dest) + n + 1`，因为该函数还会追加字符串结束符`'\0'`到`dest`后面。

下面的示例为正确的写法：

```
char dest[20] = "hello";
strncat(dest, src, sizeof(dest) - strlen(dest) - 1);
```

一定要记住给`'\0'`留下空间。

2) 使用strncpy代替strcpy, 代码如下:

```
#include <string.h>
char *strncpy(char *dest, const char *src, size_t n);
```

从src中最多复制n个字符到dest字符串中。与strncat相同的是, 当src包含n个以上的字符时, dest的空间需要为n+1, 因为该函数还会再复制一个字符串结束符'\0'。

下面的示例为正确的写法:

```
char dest[20];
strncpy(dest, src, sizeof(dest)-1);
```

3) 使用snprintf代替sprintf, 代码如下:

```
#include <stdio.h>
int snprintf(char *str, size_t size, const char *format, ...);
```

snprintf比前面两个函数strncat和strncpy更为友好, 在往str中写数据时, 最多会写入n字节, 其中已包括字符串结束符'\0'。

正确的示例代码如下:

```
char str[20];
snprintf(str, sizeof(str), "%s", dest0);
```

4) 使用fgets代替gets, 代码如下:

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
```

危险的gets函数从来不检查缓冲区的大小, 并且还是从标准输入中读取数据, 这是极其危险的行为。再大的缓存空间也无法满足永无终止的标准输入, 因此一定要使用fgets代替。

fgets最多会复制size-1字节到缓存s中, 并且会在最后一个字符后面追加'\0'。因此如果要读取标准输入, 正确的示例代码如下:

```
char str[20];
fgets(str, sizeof(str), stdin);
```

由于历史原因, 标准C库中还存在其他不安全的接口, 不过后来C库中也发展了相应的安全接口。在日常的编程中, 除非特殊情况, 都要使用安全函数来替代非安全函数的调用。

3.6.3 如何定位内存问题

前文主要介绍了如何防范和避免内存问题。但是如果程序里面真的出现了内存问题，我们又该如何定位它，如何找到根本原因呢？

工欲善其事，必先利其器。**valgrind**作为一个免费且优秀的工具包，提供了很多有用的功能，其中最有名的就是对内存问题的检测和定位。

请看下面的代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
static void mem_leak1(void)
{
    char *p = malloc(1);
}
static void mem_leak2(void)
{
    FILE *fp = fopen("test.txt", "w");
}
static void mem_overrun1(void)
{
    char *p = malloc(1);
    *(short*)p = 2;
    free(p);
}
static void mem_overrun2(void)
{
    char array[5];
    strcpy(array, "hello");
}
static void mem_double_free(void)
{
    char *p = malloc(1);
    free(p);
    free(p);
}
static void mem_free_wild_pointer(void)
{
    char *p;
    free(p);
}
int main()
{
    mem_leak1();
    mem_leak2();
    mem_overrun1();
    mem_overrun2();
    mem_double_free();
    mem_free_wild_pointer();
    return 0;
}
```

上面的代码中包含了六种常见的内存问题：

- 动态内存泄漏；
- 资源泄漏，代码中以文件描述符为例；
- 动态内存越界；
- 数组越界；

·动态内存double free;

·使用野指针。

下面来看看怎样执行valgrind来检测内存错误:

```
valgrind --track-fds=yes --leak-check=full --undef-value-errors=yes ./mem_test
```

这段代码中各项的具体含义, 可以参看valgrind--help, 其中有些option默认就是打开的, 不过笔者习惯于明确地使用option, 以示清晰。

下面来看看执行后的报告:

```
==2326== Memcheck, a memory error detector
==2326== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2326== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==2326== Command: ./mem_test
==2326==
/* 此处检测到了动态内存的越界, 提示
```

```
Invalid write*/
==2326== Invalid write of size 2
==2326== at 0x80484B4: mem_overnl (in /home/fgao/works/test/a.out)
==2326== by 0x8048553: main (in /home/fgao/works/test/a.out)
==2326== Address 0x40211f0 is 0 bytes inside a block of size 1 alloc'd
==2326== at 0x4005BDC: malloc (vg_replace_malloc.c:195)
==2326== by 0x80484AD: mem_overnl (in /home/fgao/works/test/a.out)
==2326== by 0x8048553: main (in /home/fgao/works/test/a.out)
==2326==
/* 此处检测到了
```

double free的问题, 提示

```
Invalid Free */
==2326== Invalid free() / delete / delete[]
==2326== at 0x40057F6: free (vg_replace_malloc.c:325)
==2326== by 0x8048514: mem_double_free (in /home/fgao/works/test/a.out)
==2326== by 0x804855D: main (in /home/fgao/works/test/a.out)
==2326== Address 0x4021228 is 0 bytes inside a block of size 1 free'd
==2326== at 0x40057F6: free (vg_replace_malloc.c:325)
==2326== by 0x8048509: mem_double_free (in /home/fgao/works/test/a.out)
==2326== by 0x804855D: main (in /home/fgao/works/test/a.out)
==2326==
/* 此处检测到了未初始化变量的问题
```

```
*/
==2326== Conditional jump or move depends on uninitialised value(s)
==2326== at 0x40057B6: free (vg_replace_malloc.c:325)
==2326== by 0x804853C: mem_free_wild_pointer (in /home/fgao/works/test/a.out)
==2326== by 0x8048562: main (in /home/fgao/works/test/a.out)
==2326==
/* 此处检测到了非法使用野指针
```

```
*/
==2326== Invalid free() / delete / delete[]
==2326== at 0x40057F6: free (vg_replace_malloc.c:325)
==2326== by 0x804853C: mem_free_wild_pointer (in /home/fgao/works/test/a.out)
==2326== by 0x8048562: main (in /home/fgao/works/test/a.out)
==2326== Address 0x4021228 is 0 bytes inside a block of size 1 free'd
==2326== at 0x40057F6: free (vg_replace_malloc.c:325)
==2326== by 0x8048509: mem_double_free (in /home/fgao/works/test/a.out)
```

```
==2326== by 0x804855D: main (in /home/fgao/works/test/a.out)
==2326==
==2326==
/*此处检测到了文件指针资源的泄漏，下面提示说有
```

4个文件描述符在退出时仍是打开的描述

符

0、

1、

2无须关心，通过报告，可以发现程序中自己明确打开的文件描述符没有关闭

```
*/
==2326== FILE DESCRIPTORS: 4 open at exit.
==2326== Open file descriptor 3: test.txt
==2326== at 0x68D613: __open_nocancel (in /lib/libc-2.12.so)
==2326== by 0x61F8EC: __fopen_internal (in /lib/libc-2.12.so)
==2326== by 0x61F94B: fopen@@GLIBC_2.1 (in /lib/libc-2.12.so)
==2326== by 0x8048496: mem_leak2 (in /home/fgao/works/test/a.out)
==2326== by 0x804854E: main (in /home/fgao/works/test/a.out)
==2326==
==2326== Open file descriptor 2: /dev/pts/4
==2326== <inherited from parent>
==2326==
==2326== Open file descriptor 1: /dev/pts/4
==2326== <inherited from parent>
==2326==
==2326== Open file descriptor 0: /dev/pts/4
==2326== <inherited from parent>
==2326==
==2326==
/* 堆信息的总结：一共调用了
```

4次

alloc.

4次

free. 之所以正好相等，是因为上面有一个函数少了

free, 有一个函数正好又多了一个

```
free */
==2326== HEAP SUMMARY:
==2326== in use at exit: 353 bytes in 2 blocks
==2326== total heap usage: 4 allocs, 4 frees, 355 bytes allocated
==2326==
/* 检测到一字节的内存泄漏
```

```
*/
==2326== 1 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2326== at 0x4005BDC: malloc (vg_replace_malloc.c:195)
==2326== by 0x8048475: mem_leak1 (in /home/fgao/works/test/a.out)
==2326== by 0x8048549: main (in /home/fgao/works/test/a.out)
==2326==
/* 内存泄漏的总结

*/
==2326== LEAK SUMMARY:
==2326== definitely lost: 1 bytes in 1 blocks
==2326== indirectly lost: 0 bytes in 0 blocks
==2326== possibly lost: 0 bytes in 0 blocks
==2326== still reachable: 352 bytes in 1 blocks
==2326== suppressed: 0 bytes in 0 blocks
==2326== Reachable blocks (those to which a pointer was found) are not shown.
==2326== To see them, rerun with: --leak-check=full --show-reachable=yes
==2326==
==2326== For counts of detected and suppressed errors, rerun with: -v
==2326== Use --track-origins=yes to see where uninitialised values come from
==2326== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 12 from 8)
```

这只是一个简单的示例程序，即使没有**valgrind**，我们也可以很轻易地发现问题。但是在真实的项目中，当代码量达到万行、十万行甚至百万行时，由于申请的内存可能不是在一个地方被使用，它不可避免地会被传来传去。这时，如果只是靠**review**代码来检查问题，可能很难找到根本原因，而使用**valgrind**则可以很容易地发现问题所在。

3.7 “长跳转”longjmp

C语言中的goto语句由于可以直接跳转到函数中的任意一行，因此是一个颇受争议的语句。有人认为它给代码带来了混乱，有人则认为适当地使用goto语句可以让代码更简洁、清晰——比如内核代码中就充斥着goto语句的使用。关于这点，仁者见仁，智者见智吧。

goto语句已经引发了这么大的争议，而C库还提供了另外一组接口，用于实现“长跳转”。对比goto语句只能在函数内部的“短跳转”，longjmp可以实现跨函数的“长跳转”。下面我们来详细看看longjmp的使用方法。

3.7.1 setjmp与longjmp的使用

我们先来看看setjmp的代码：

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

setjmp用于保存当前栈的上下文，将其保存到参数env中。若返回0值，则为setjmp直接返回的结果；若返回非0值，则为从longjmp恢复栈空间时返回的结果。

longjmp用于将上下文恢复至env保存的状态，参数val用于作为恢复点setjmp的返回值。一般情况下，保存的jmp_buf env为全局变量。跳转一次后，保存的env上下文环境就会失效。请看下面的示例：

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
static jmp_buf g_stack_env;
static void func1(void);
static void func2(void);
int main(void)
{
    if (0 == setjmp(g_stack_env)) {
        printf("Normal flow\n");
        func1();} else {
        printf("Longjump flow\n");
    }
    return 0;
}
static void func1(void)
{
    printf("Enter func1\n");
    func2();
}
static void func2(void)
{
    printf("Enter func2\n");
    longjmp(g_stack_env, 1);
    printf("Leave func2\n");
}
```

其输出结果为：

```
Normal flow
Enter func1
Enter func2
Longjump flow
```

在main函数中，使用setjmp将当前的栈环境保存到g_stack_env中，然后调用func1->func2，在func2中，使用longjmp来恢复保存的栈环境g_stack_env，从而完成“长跳转”。

3.7.2 “长跳转”的实现机制

setjmp和longjmp分别用于保存和恢复栈的上下文，来实现长跳转。而栈的实现肯定是与平台相关的，因此setjmp和longjmp的实现也是与平台相关的。

先看一下struct jmp_buf的定义：

```
/* Calling environment, plus possibly a saved signal mask. */
struct __jmp_buf_tag
{
    /* NOTE: The machine-dependent definitions of `__sigsetjmp'
     * assume that a `jmp_buf' begins with a `__jmp_buf' and that `__mask_was_saved' follows it. Do not move these members or add others before it. */
    __jmp_buf __jmpbuf; /* Calling environment. */
    int __mask_was_saved; /* Saved the signal mask? */
    __sigset_t __saved_mask; /* Saved signal mask. */
};
typedef struct __jmp_buf_tag jmp_buf[1];
```

x86平台的__jmp_buf的定义为：

```
# if WORDSIZE == 64
typedef long int __jmp_buf[8];
# elif defined x86_64
typedef long long int __jmp_buf[8];
# else
typedef int __jmp_buf[6];
# endif
```

x86平台的setjmp和longjmp的实现均位于glibc-2.17/sysdeps/i386/setjmp.S中。

```
ENTRY (BP_SYM (__sigsetjmp))
ENTER
/* 将

jmpbuf的地址赋给

eax */
movl JMPBUF(%esp), %eax
CHECK_BOUNDS_BOTH_WIDE (%eax, JMPBUF(%esp), $JB_SIZE)
/* 保存寄存器

*/
movl %ebx, (JB_BX*4)(%eax)
movl %esi, (JB_SI*4)(%eax)
movl %edi, (JB_DI*4)(%eax)
leal JMPBUF(%esp), %ecx /* Save SP as it will be after we return. */
#ifdef PTR_MANGLE
PTR_MANGLE (%ecx)
#endif
movl %ecx, (JB_SP*4)(%eax)
movl PCOFF(%esp), %ecx /* Save PC we are returning to now. */
LIBC_PROBE (setjmp, 3, 4@%eax, -4@SIGMSK(%esp), 4@%ecx)
#ifdef PTR_MANGLE
PTR_MANGLE (%ecx)
#endif
movl %ecx, (JB_PC*4)(%eax)
LEAVE /* pop frame pointer to prepare for tail-call. */
movl %ebp, (JB_BP*4)(%eax) /* Save caller's frame pointer. */
#ifdef NOT_IN_libc && defined IS_IN_rtdl
/* In ld.so we never save the signal mask. */
xorl %eax, %eax
ret
#else
/* Make a tail call to __sigjmp_save; it takes the same args. */
jmp __sigjmp_save
#endif
END (BP_SYM (__sigsetjmp))
```

上面的汇编代码，主要是将寄存器EBX、ESI、EDI、ESP、PC和EBP寄存器保存到jmp_buf中。回想前面__jmp_buf的定义，它在x8632位平台上是大小为6的int型数组，正好用于保存这6个寄存器。



提示 细心的读者会发现这里的汇编是__sigsetjmp的实现，而不是setjmp的实现。那是因为是在glibc库中，setjmp是调用__sigsetjmp来实现的。

看完了__sigsetjmp的实现，自然就轮到longjmp了：

```
ENTRY (__longjmp)
movl 4(%esp), %ecx /* User's jmp_buf in %ecx. */
movl 8(%esp), %eax /* Second argument is return value. */
/* Save the return address now. */
movl (JB_PC*4)(%ecx), %edx
LIBC_PROBE (longjmp, 3, 4@%ecx, -4@%eax, 4@%edx)
/* 恢复保存的寄存器
```



```
*/
movl (JB_BX*4)(%ecx), %ebx
movl (JB_SI*4)(%ecx), %esi
movl (JB_DI*4)(%ecx), %edi
movl (JB_BP*4)(%ecx), %ebp
movl (JB_SP*4)(%ecx), %esp
LIBC_PROBE (longjmp_target, 3, 4@%ecx, -4@%ecx, 4@%edx)
/* Jump to saved PC. */
jmp *%edx
END (__longjmp)
```

`setjmp`保存寄存器的内容，`longjmp`自然是恢复寄存器的内容。上面的代码很简单，把寄存器PC、EBX、ESI、EDI、EBP和ESP的内容恢复后，将第二个参数`val`保存到EAX中，最后跳转到恢复的PC寄存器处——也就是`setjmp`的下一条指令的位置。

3.7.3 “长跳转”的陷阱

从3.7.2节对setjmp和longjmp实现的分析中，我们可以发现，setjmp和longjmp的实现原理就是对与栈相关的寄存器的保存与恢复。那么，变量的情况又是什么样的呢？对于全局变量和static变量来说，由于它们都不是保存在栈上的，所以在longjmp跳转后，其值不会改变。局部变量的情况又如何呢？

longjmp的man手册给出了如下说明：

当满足以下条件时，局部变量的值是不能确定的：

- 它们是调用setjmp所在函数的局部变量。
- 其值在setjmp和longjmp之间有变化。
- 它们没有被声明为volatile变量。

我们来做一个试验：

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
static jmp_buf g_stack_env;
static void func1(int *a, int *b, int *c);
int main(void)
{
    int a = 1;
    int b = 2;
    int c = 3;
    int ret = setjmp(g_stack_env);
    if (0 == ret) {
        printf("Normal flow\n");
        printf("a = %d, b = %d, c = %d\n", a, b, c);
        func1(&a, &b, &c);
    } else {
        printf("Longjump flow\n");
        printf("a = %d, b = %d, c = %d\n", a, b, c);
    }
    return 0;
}
static void func1(int *a, int *b, int *c)
{
    printf("Enter func1\n");
    ++(*a);
    ++(*b);
    ++(*c);
    printf("func1: a = %d, b = %d, c = %d\n", *a, *b, *c);
    longjmp(g_stack_env, 1);
    printf("Leave func1\n");
}
```

然后编译运行：

```
[fgao@ubuntu chapter3]#gcc 4_7_3_longjmp_var.c -Wall
[fgao@ubuntu chapter3]#./a.out
Normal flow
a = 1, b = 2, c = 3
Enter func1
func1: a = 2, b = 3, c = 4
Longjump flow
a = 2, b = 3, c = 4
```

从结果上看，变量a、b、c的值均没有被恢复。这点符合我们的预期，毕竟longjmp只是恢复了6个

寄存器的内容。

然而当我们加上编译选项-O2以后，结果就完全不同了。

```
[fgao@ubuntu chapter3]#gcc 4_7_3_longjmp_var.c -Wall -O2
[fgao@ubuntu chapter3]#./a.out
Normal flow
a = 1, b = 2, c = 3
Enter func1
func1: a = 2, b = 3, c = 4
Longjump flow
a = 1, b = 2, c = 3
```

在longjmp跳转以后，a、b和c的值仍然是原来的值。

除了上面这个缺陷以外，如果我们的思维再开阔些，还能发现由longjmp实现原理引发的其他缺陷。比如因为它不能处理局部变量的问题，因此在C++中局部变量的析构肯定也是有问题的。

请看下面的示例：

```
#include <setjmp.h>
#include <iostream>
using namespace std;
static jmp_buf g_stack_env;
static void func1(void);
class Test {
public:
    Test() {
        cout << "Constructor" << endl;
    }
    ~Test() {
        cout << "Destructor" << endl;
    }
};
int main(void)
{
    int ret = setjmp(g_stack_env);
    if (0 == ret) {
        cout << "Normal flow" << endl;
        func1();
    } else {
        cout << "Longjump flow" << endl;
    }
}
```

其输出结果为：

```
[fgao@ubuntu chapter3]#g++ 4_7_3_longjmp_destructor.cpp -Wall
[fgao@ubuntu chapter3]#./a.out
Normal flow
Enter func1
Constructor
Longjump flow
```

之所以Test的析构函数没有被调用，是因为longjmp是glibc库中的函数，它直接恢复了栈的上下文，因此程序不会调用Test的析构函数。

第4章 进程控制：进程的一生

进程是操作系统的一个核心概念。每个进程都有自己唯一的标识：进程ID，也有自己的生命周期。一个典型的进程的生命周期如图4-1所示。

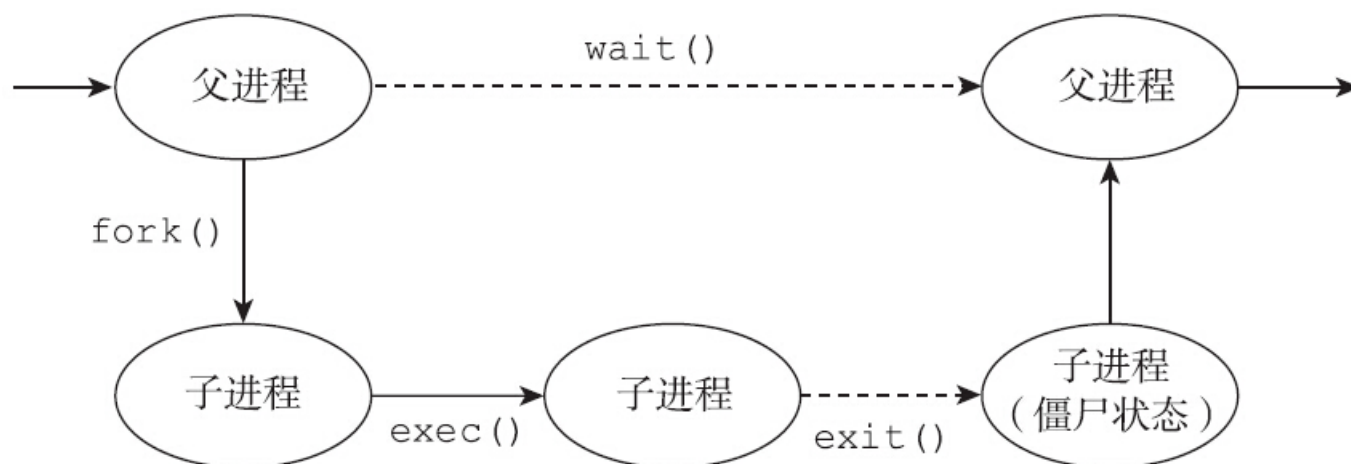


图4-1 进程的生命周期

本章将会介绍进程ID、进程的层次，以及进程生命周期内的各个阶段。

4.1 进程ID

Linux下每个进程都会有一个非负整数表示的唯一进程ID，简称pid。Linux提供了getpid函数来获取进程的pid，同时还提供了getppid函数来获取父进程的pid，相关接口定义如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

每个进程都有自己的父进程，父进程又会有自己的父进程，最终都会追溯到1号进程即init进程。这就决定了操作系统上所有的进程必然会组成树状结构，就像一个家族的家谱一样。可以通过pstree的命令来查看进程的家族树。

procfs文件系统会在/proc下为每个进程创建一个目录，名字是该进程的pid。目录下有很多文件，用于记录进程的运行情况和统计信息等，如下所示：

ll /proc总用量				
0				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 1				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 10				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 100				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 101				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 102				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 103				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 1039				
dr-xr-xr-x	9	root	root	0 4月
1 06:56 104				

因为进程有创建，也有终止，所以`/proc/`下记录进程信息的目录（以及目录下的文件）也会发生变化。

操作系统必须保证在任意时刻都不能出现两个进程有相同pid的情况。虽然进程ID是唯一的，但是进程ID可以重用。进程退出以后，其进程ID还可以再次分配给其他的进程使用。那么问题就来了，内核是如何分配进程ID的？

Linux分配进程ID的算法不同于给进程分配文件描述符的最小可用算法，它采用了延迟重用的算法，即分配给新创建进程的ID尽量不与最近终止进程的ID重复，这样就可以防止将新创建的进程误判为使用相同进程ID的已经退出的进程。

那么如何实现延迟重用呢？内核采用的方法如下：

1) 位图记录进程ID的分配情况（0为可用，1为已占用）。

2) 将上次分配的进程ID记录到`last_pid`中，分配进程ID时，从`last_pid+1`开始找起，从位图中寻找可用的ID。

3) 如果找到位图集合的最后一位仍不可用，则回滚到位图集合的起始位置，从头开始找。

既然是位图记录进程ID的分配情况，那么位图的大小就必须要考虑周全。位图的大小直接决定了系统允许同时存在的进程的最大个数，这个最大个数在系统中称为`pid_max`。

上面的第3步提到，回绕到位图集合的起始位置，从头寻找可用的进程ID。事实上，严格说来，这种说法并不正确，回绕时并不是从0开始找起，而是从300开始找起。内核在`kernel/pid.c`文件中定义了`RESERVED_PIDS`，其值是300，300以下的pid会被系统占用，而不能分配给用户进程：

```
define RESERVED_PIDS      300
int pid_max = PID_MAX_DEFAULT;
```

Linux系统下可以通过`procfs`或`sysctl`命令来查看`pid_max`的值：

```
manu@manu-rush:~$ cat /proc/sys/kernel/pid_max
131072
manu@manu-rush:~$ sysctl kernel.pid_max
kernel.pid_max = 131072
```

其实，此上限值是可以调整的，系统管理员可以通过如下方法来修改此上限值：

```
root@manu-rush:~# sysctl -w kernel.pid_max=4194304
kernel.pid_max = 4194304
```

但是内核自己也设置了硬上限，如果尝试将pid_max的值设成一个大于硬上限的值就会失败，如下所示：

```
root@manu-rush:~# sysctl -w kernel.pid_max=4194305
error: "Invalid argument" setting key "kernel.pid_max"
```

从上面的操作可以看出，Linux系统将系统进程数的硬上限设置为4194304（4M）。内核又是如何决定系统进程个数的硬上限的呢？对此，内核定义了如下的宏：

```
#define PID_MAX_LIMIT (CONFIG_BASE_SMALL ? PAGE_SIZE * 8 : \
    (sizeof(long) > 4 ? 4 * 1024 * 1024 : PID_MAX_DEFAULT))
```

从上面代码中可以看出决定系统进程个数硬上限的逻辑为：

- 如果选择了CONFIG_BASE_SMALL编译选项，则为页面（PAGE_SIZE）的位数。
- 如果选择了CONFIG_BASE_FULL编译选项，那么：
 - 对于32位系统，系统进程个数硬上限为32768（即32K）。
 - 对于64位系统，系统进程个数硬上限为4194304（即4M）。

通过上面的讨论可以看出，在64位系统中，系统容许创建的进程的个数超过了400万，这个数字是相当庞大的，足够应用层使用。

对于单线程的程序，进程ID比较好理解，就是唯一标识进程的数字。对于多线程的程序，每一个线程调用getpid函数，其返回值都是一样的，即进程的ID。

4.2 进程的层次

每个进程都有父进程，父进程也有父进程，这就形成了一个以init进程为根的家族树。除此以外，进程还有其他层次关系：进程、进程组和会话。

进程组和会话在进程之间形成了两级的层次：进程组是一组相关进程的集合，会话是一组相关进程组的集合。用人来打比方，会话如同一个公司，进程组如同公司里的部门，进程则如同部门里的员工。尽管每个员工都有父亲，但是不影响员工同时属于某个公司中的某个部门。

这样说来，一个进程会有如下ID：

- PID：进程的唯一标识。对于多线程的进程而言，所有线程调用getpid函数会返回相同的值。

- PGID：进程组ID。每个进程都会有进程组ID，表示该进程所属的进程组。默认情况下新创建的进程会继承父进程的进程组ID。

- SID：会话ID。每个进程也都有会话ID。默认情况下，新创建的进程会继承父进程的会话ID。

可以调用如下指令来查看所有进程的层次关系：

```
ps -ejH
ps axjf
```

对于进程而言，可以通过如下函数调用来获取其进程组ID和会话ID。

```
#include <unistd.h>
pid_t getpgrp(void);
pid_t getsid(pid_t pid);
```

前面提到过，新进程默认继承父进程的进程组ID和会话ID，如果都是默认情况的话，那么追根溯源可知，所有的进程应该有共同的进程组ID和会话ID。但是调用ps axjf可以看到，实际情况并非如此，系统中存在很多不同的会话，每个会话下也有不同的进程组。

为何会如此呢？

就像家族企业一样，如果从创业之初，所有家族成员都墨守成规，循规蹈矩，默认情况下，就只会会有一个公司、一个部门。但是也有些“叛逆”的子弟，愿意为家族公司开疆拓土，愿意成立新的部门。这些新的部门就是新创建的进程组。如果有子弟“离经叛道”，甚至不愿意呆在家族公司里，他别开天地，另创了一个公司，那这个新公司就是新创建的会话组。由此可见，系统必须要有改变和设置进程组ID和会话ID的函数接口，否则，系统中只会存在一个会话、一个进程组。

进程组和会话是为了支持shell作业控制而引入的概念。

当有新的用户登录Linux时，登录进程会为用户创建一个会话。用户的登录shell就是会话的首进程。会话的首进程ID会作为整个会话的ID。会话是一个或多个进程组的集合，囊括了登录用户的所有活动。

在登录shell时，用户可能会使用管道，让多个进程互相配合完成一项工作，这一组进程属于同一个进程组。

当用户通过SSH客户端工具（putty、xshell等）连入Linux时，与上述登录的情景是类似的。

4.2.1 进程组

修改进程组ID的接口如下：

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

这个函数的含义是，找到进程ID为pid的进程，将其进程组ID修改为pgid，如果pid的值为0，则表示要修改调用进程的进程组ID。该接口一般用来创建一个新的进程组。

下面三个接口含义一致，都是创立新的进程组，并且指定的进程会成为进程组的首进程。如果参数pid和pgid的值不匹配，那么setpgid函数会将一个进程从原来所属的进程组迁移到pgid对应的进程组。

```
setpgid(0,0)
setpgid(getpid(),0)
setpgid(getpid(),getpid())
```

setpgid函数有很多限制：

- pid参数必须指定为调用setpgid函数的进程或其子进程，不能随意修改不相关进程的进程组ID，如果违反这条规则，则返回-1，并置errno为ESRCH。

- pid参数可以指定调用进程的子进程，但是子进程如果已经执行了exec函数，则不能修改子进程的进程组ID。如果违反这条规则，则返回-1，并置errno为EACCESS。

- 在进程组间移动，调用进程，pid指定的进程及目标进程组必须在同一个会话之内。这个比较好理解，不加入公司（会话），就无法加入公司下属的部门（进程组），否则就是部门要造反的节奏。如果违反这条规则，则返回-1，并置errno为EPERM。

- pid指定的进程，不能是会话首进程。如果违反这条规则，则返回-1，并置errno为EPERM。

有了创建进程组的接口，新创建的进程组就不必继承父进程的进程组ID了。最常见的创建进程组的场景就是在shell中执行管道命令，代码如下：

```
cmd1 | cmd2 | cmd3
```

下面用一个最简单的命令来说明，其进程之间的关系如图4-2所示。

```
ps ax|grep nfsd
```

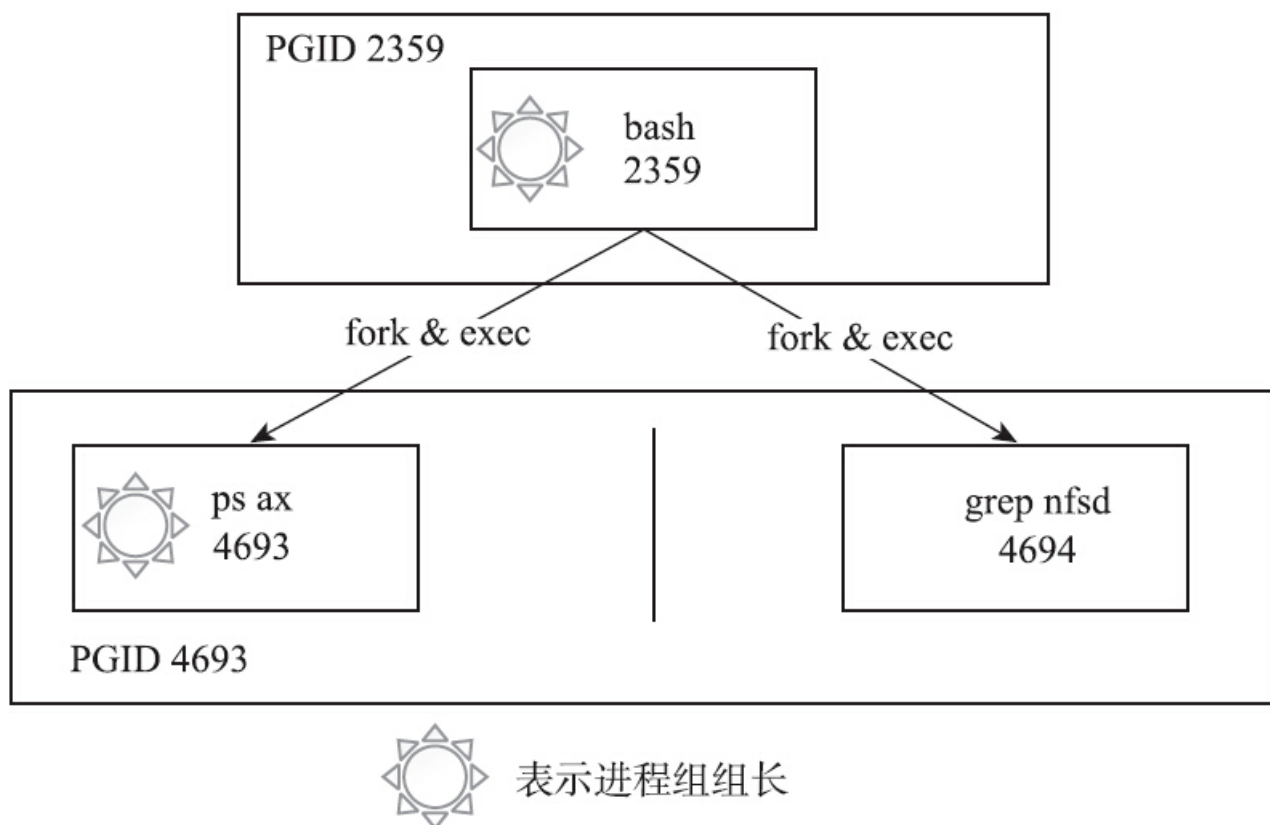


图4-2 进程组和进程的关系

ps进程和grep进程都是bash创建的子进程，两者通过管道协同完成一项工作，它们隶属于同一个进程组，其中ps进程是进程组的组长。

进程组的概念并不难理解，可以将人与人之间的关系做类比。一起工作的同事，自然比毫不相干的路人更加亲近。shell中协同工作的进程属于同一个进程组，就如同协同工作的人属于同一个部门一样。

引入了进程组的概念，可以更方便地管理这一组进程了。比如这项工作放弃了，不必向每个进程一一发送信号，可以直接将信号发送给进程组，进程组内的所有进程都会收到该信号。

前文曾提到过，子进程一旦执行exec，父进程就无法调用setpgid函数来设置子进程的进程组ID了，这条规则会影响shell的作业控制。出于保险的考虑，一般父进程在调用fork创建子进程后，会调用setpgid函数设置子进程的进程组ID，同时子进程也要调用setpgid函数来设置自身的进程组ID。这两次调用有一次是多余的，但是这样做能够保证无论是父进程先执行，还是子进程先执行，子进程一定已经进入了指定的进程组中。由于fork之后，父子进程的执行顺序是不确定的，因此如果不这样做，就会造成在一定的时间窗口内，无法确定子进程是否进入了相应的进程组。

可以通过跟踪bash进程的系统调用来证明这一点，下面的2258进程是bash，我们在该bash上执行sleep 200，在执行之前，在另一个终端用strace跟踪bash的系统调用，可以看到，父进程和子进程都执行了一遍setpgid函数，代码如下所示：

```
manu@manu-hacks:~$ sudo strace -f -p 2258
Process 2258 attached
```

```
....
```

```
/*父进程调用
```

```
setpgid函数
```

```
*/
[pid 2258] setpgid(2509, 2509 <unfinished ...>...
```

```
/*子进程调用
```

```
setpgid函数
```

```
*/
[pid 2509] setpgid(2509, 2509 <unfinished ...>...
```

```
/*子进程执行
```

```
execve*/
[pid 2509] execve("/bin/sleep", ["sleep", "200"], [/* 31 vars */]) = 0...
```

用户在**shell**中可以同时执行多个命令。对于耗时很久的命令（如编译大型工程），用户不必傻傻等待命令运行完毕才执行下一个命令。用户在执行命令时，可以在命令的结尾添加“&”符号，表示将命令放入后台执行。这样该命令对应的进程组即为后台进程组。在任意时刻，可能同时存在多个后台进程组，但是不管什么时候都只能有一个前台进程组。只有在前台进程组中进程才能在控制终端读取输入。当用户在终端输入信号生成终端字符（如ctrl+c、ctrl+z、ctr+\等）时，对应的信号只会发送给前台进程组。

shell中可以存在多个进程组，无论是前台进程组还是后台进程组，它们或多或少存在一定的联系，为了更好地控制这些进程组（或者称为作业），系统引入了会话的概念。会话的意义在于将很多的工作囊括在一个终端，选取其中一个作为前台来直接接收终端的输入及信号，其他的工作则放在后台执行。

4.2.2 会话

会话是一个或多个进程组的集合，以用户登录系统为例，可能存在如图4-3所示的情况。

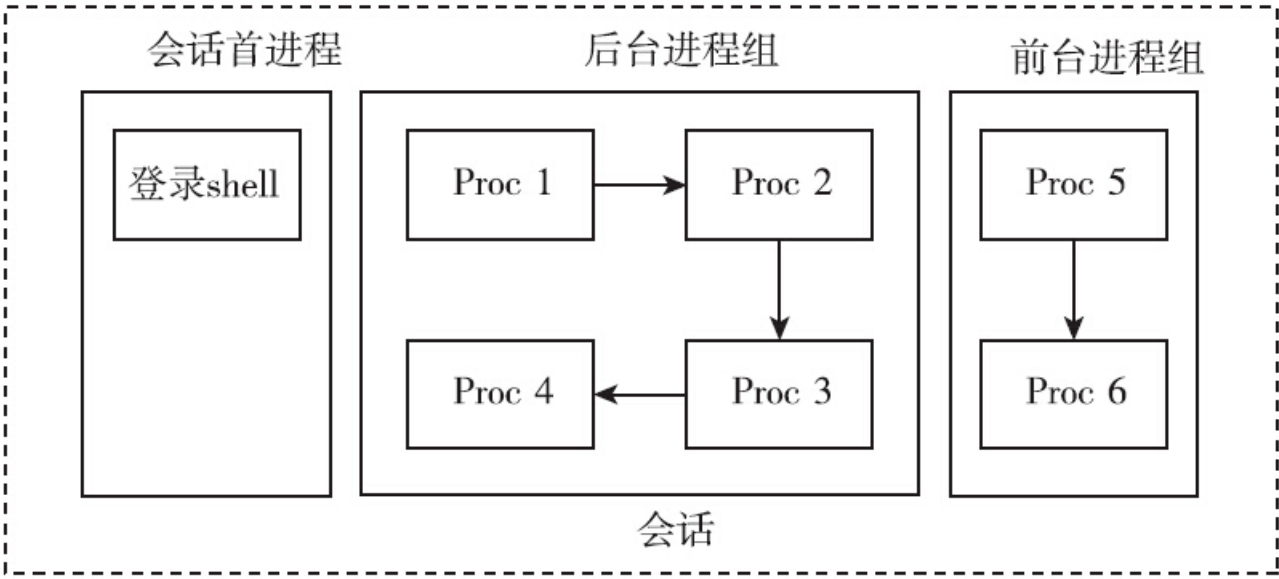


图4-3 进程组与会话的关系

系统提供setsid函数来创建会话，其接口定义如下：

```
#include <unistd.h>
pid_t setsid(void);
```

如果这个函数的调用进程不是进程组组长，那么调用该函数会发生以下事情：

- 1) 创建一个新会话，会话ID等于进程ID，调用进程成为会话的首进程。
- 2) 创建一个进程组，进程组ID等于进程ID，调用进程成为进程组的组长。
- 3) 该进程没有控制终端，如果调用setsid前，该进程有控制终端，这种联系就会断掉。

调用setsid函数的进程不能是进程组的组长，否则调用会失败，返回-1，并置errno为EPERM。

这个限制是比较合理的。如果允许进程组组长迁移到新的会话，而进程组的其他成员仍然在老的会话中，那么，就会出现同一个进程组的进程分属不同的会话之中的情况，这就破坏了进程组和会话的严格的层次关系了。

Linux提供了setsid命令，可以在新的会话中执行命令，通过该命令可以很容易地验证上面提到的三点：

```
manu@manu-hacks:~$ setsid sleep 100
manu@manu-hacks:~$ ps ajxf
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND...
1	4469	4469	4469	?	-1	Ss	1000	0:00	sleep 100

从输出中可以看出，系统创建了新的会话**4469**，新的会话下又创建了新的进程组，会话**ID**和进程组**ID**都等于进程**ID**，而该进程已经不再拥有任何控制终端了（TTY对应的值为“?”表示进程没有控制终端）。

常用的调用**setsid**函数的场景是**login**和**shell**。除此以外创建**daemon**进程也要调用**setsid**函数。

4.3 进程的创建之fork（）

Linux系统下，进程可以调用fork函数来创建新的进程。调用进程为父进程，被创建的进程为子进程。

fork函数的接口定义如下：

```
#include <unistd.h>
pid_t fork(void);
```

与普通函数不同，fork函数会返回两次。一般说来，创建两个完全相同的进程并没有太多的价值。大部分情况下，父子进程会执行不同的代码分支。fork函数的返回值就成了区分父子进程的关键。fork函数向子进程返回0，并将子进程的进程ID返给父进程。当然了，如果fork失败，该函数则返回-1，并设置errno。

常见的出错情景如表4-1所示。

表4-1 fork函数可能的errno

errno	说 明
EAGAIN	超出了用户容许创建的进程上限，也可能是超出了系统容许的进程个数的上限
ENOMEM	无法分配相应的内核结构，内存紧张的情况下，可能发生该错误
ENOSYS	平台不支持 fork

所以一般而言，调用fork的程序，大多会如此处理：

```
ret = fork();
if(ret == 0)
{
    ...

    // 此处是子进程的代码分支

}
else if(ret > 0)
{
    ...

    // 此处是父进程的代码分支

}
else
{
    ...

    // fork失败，执行

    error handle
}
```



注意 fork可能失败。检查返回值进行正确的出错处理，是一个非常重要的习惯。设想如果fork返回-1，而程序没有判断返回值，直接将-1当成子进程的进程号，那么后面的代码执行kill（child_pid，9）就相当于执行kill（-1，9）。这会发生什么？后果是惨重的，它将杀死除了init以外的所有进程，只要它有权限。读者可以通过man 2 kill来查看kill（-1，9）的含义。

fork之后，对于父子进程，谁先获得CPU资源，而率先运行呢？

从内核2.6.32开始，在默认情况下，父进程将成为fork之后优先调度的对象。采取这种策略的原因是：fork之后，父进程在CPU中处于活跃的状态，并且其内存管理信息也被置于硬件内存管理单元的转译后备缓冲器（TLB），所以先调度父进程能提升性能。

从2.6.24起，Linux采用完全公平调度（Completely Fair Scheduler，CFS）。用户创建的普通进程，都采用CFS调度策略。对于CFS调度策略，procfs提供了如下控制选项：

```
/proc/sys/kernel/sched_child_runs_first
```

该值默认是0，表示父进程优先获得调度。如果将该值改成1，那么子进程会优先获得调度。

POSIX标准和Linux都没有保证会优先调度父进程。因此在应用中，决不能对父子进程的执行顺序做任何的假设。如果确实需要某一特定执行的顺序，那么需要使用进程间同步的手段。

4.3.1 fork之后父子进程的内存关系

fork之后的子进程完全拷贝了父进程的地址空间，包括栈、堆、代码段等。通过下面的示例代码，我们一起来查看父子进程的内存关系：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <wait.h>
int g_int = 1;
int main()
{
    int local_int = 1;
    int *malloc_int = malloc(sizeof(int));
    *malloc_int = 1;
    pid_t pid = fork();
    if(pid == 0) /*子进程*/

    /*
    {
        local_int = 0;
        g_int = 0;
        *malloc_int = 0;
        fprintf(stderr, "[CHILD ] child change local global malloc value to 0\n");
        free(malloc_int);
        sleep(10);
        fprintf(stderr, "[CHILD ] child exit\n");
        exit(0);
    }
    else if(pid < 0)
    {
        printf("fork failed (%s)",strerror(errno));
        return 1;
    }
    fprintf(stderr, "[PARENT] wait child exit\n");
    waitpid(pid, NULL, 0);
    fprintf(stderr, "[PARENT] child have exit\n");
    printf("[PARENT] g_int = %d\n", g_int);
    printf("[PARENT] local_int = %d\n", local_int);
    printf("[PARENT] malloc_int = %d\n", *malloc_int);
    free(malloc_int);
    return 0;
}
```

这里刻意定义了三个变量，一个是位于数据段的全局变量，一个是位于栈上的局部变量，还有一个是通过malloc动态分配位于堆上的变量，三者的初始值都是1。然后调用fork创建子进程，子进程将三个变量的值都改成了0。

按照fork的语义，子进程完全拷贝了父进程的数据段、栈和堆上的内存，如果父子进程对相应的数据进行修改，那么两个进程是并行不悖、互不影响的。因此，在上面示例代码中，尽管子进程将三个变量的值都改成了0，对父进程而言这三个值都没有变化，仍然是1，代码的输出也证实了这一点。

```
[PARENT] wait child exit
[CHILD ] child change local global malloc value to 0
[CHILD ] child exit
[PARENT] child have exit
[PARENT] g_int = 1
[PARENT] local_int = 1
[PARENT] malloc_int = 1
```

前文提到过，子进程和父进程执行一模一样的代码的情形比较少见。Linux提供了execve系统调用，构建在该系统调用之上，glibc提供了exec系列函数。这个系列函数会丢弃现存的程序代码段，并构建新的数据段、栈及堆。调用fork之后，子进程几乎总是通过调用exec系列函数，来执行新的程序。

在这种背景下，fork时子进程完全拷贝父进程的数据段、栈和堆的做法是不明智的，因为接下来的exec系列函数会毫不留情地抛弃刚刚辛苦拷贝的内存。为了解决这个问题，Linux引入了写时拷贝（copy-on-write）的技术。

写时拷贝是指子进程的页表项指向与父进程相同的物理内存页，这样只拷贝父进程的页表项就可以了，当然要把这些页面标记成只读（如图4-4所示）。如果父子进程都不修改内存的内容，大家便相安无事，共用一份物理内存页。但是一旦父子进程中有任何一方尝试修改，就会引发缺页异常（page fault）。此时，内核会尝试为该页面创建一个新的物理页面，并将内容真正地复制到新的物理页面中，让父子进程真正地各自拥有自己的物理内存页，然后将页表中相应的表项标记为可写。

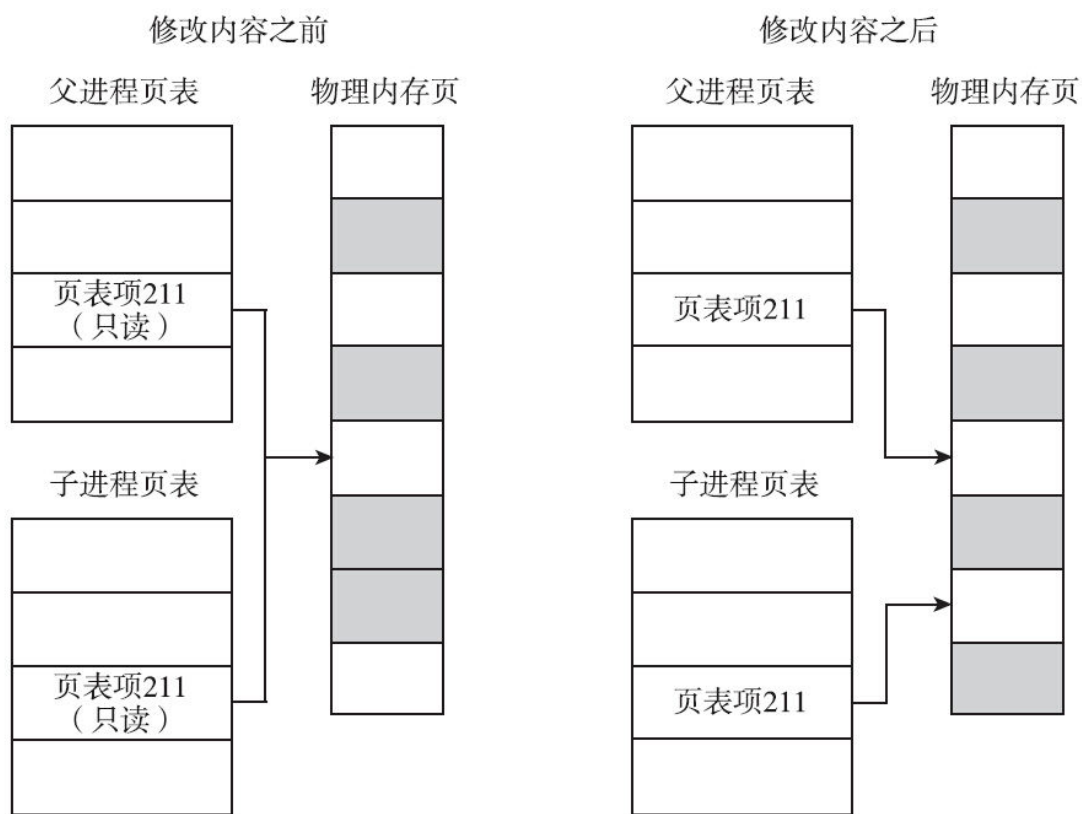


图4-4 写时拷贝

从上面的描述可以看出，对于没有修改的页面，内核并没有真正地复制物理内存页，仅仅是复制了父进程的页表。这种机制的引入提升了fork的性能，从而使内核可以快速地创建一个新的进程。

从内核代码层面来讲，其调用关系如图4-5所示。

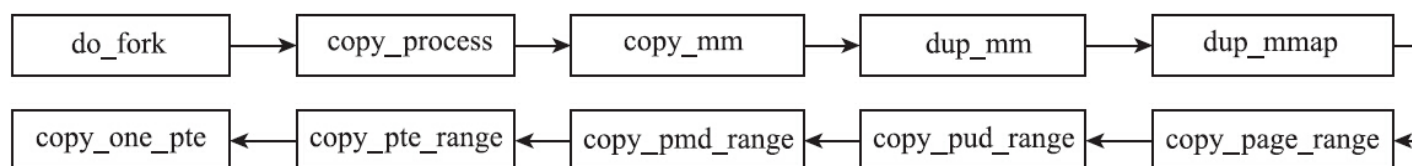


图4-5 fork复制内核页表流程

Linux的内存管理使用的是四级页表，如图4-6所示，看了四级页表的名字，也就不难推测图4-5中那些函数的作用了。

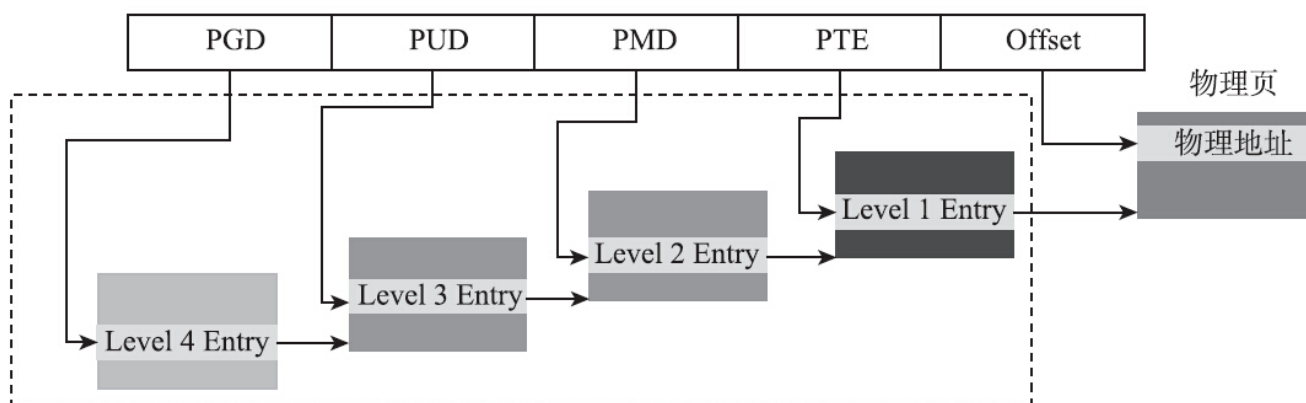


图4-6 页表的复制示意图

在最后的copy_one_pte函数中有如下代码：

```
/*如果是写时拷贝，那么无论是初始页表，还是拷贝的页表，都设置了写保护
```

*后面无论父子进程，修改页表对应位置的内存时，都会触发

```
page fault
*/
if (is_cow_mapping(vm_flags)) {
    ptep_set_wrprotect(src_mm, addr, src_pte);
    pte = pte_wrprotect(pte);
}
```

该代码将页表设置成写保护，父子进程中任意一个进程尝试修改写保护的页面时，都会引发缺页中断，内核会走向do_wp_page函数，该函数会负责创建副本，即真正的拷贝。

写时拷贝技术极大地提升了fork的性能，在一定程度上让vfork成为了鸡肋。

4.3.2 fork之后父子进程与文件的关系

执行fork函数，内核会复制父进程所有的文件描述符。对于父进程打开的所有文件，子进程也是可以操作的。那么父子进程同时操作同一个文件是并行不悖的，还是互相影响的呢？

下面通过对一个例子的讨论来说明这个问题。read函数并没有将偏移量作为参数传入，但是每次调用read函数或write函数时，却能够接着上次读写的位置继续读写。原因是内核已经将偏移量的信息记录在与文件描述符相关的数据结构里了。那么问题来了，父子进程是共用一个文件偏移量还是各有各的文件偏移量呢？

```
/*read 和  
  
write 都没有将  
  
pos信息作为入参  
  
*/  
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t count);
```

我们用事实说话，请看下面的例子：

```
#include <stdio.h>  
#include <string.h>  
#include <strings.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <errno.h>  
#define INFILE "./in.txt"  
#define OUTFILE "./out.txt"  
#define MODE S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH  
int main(void)  
{  
    int fd_in, fd_out;  
    char buf[1024];  
    memset(buf, 0, 1024);  
    fd_in = open(INFILE, O_RDONLY);  
    if (fd_in < 0)  
    {  
        fprintf(stderr, "failed to open %s, reason(%s)\n",  
INFILE, strerror(errno));  
        return 1;  
    }  
    fd_out = open(OUTFILE, O_WRONLY | O_CREAT | O_TRUNC, MODE);  
    if (fd_out < 0)  
    {  
        fprintf(stderr, "failed to open %s, reason(%s)\n", OUTFILE, strerror(errno));  
        return 1;  
    }  
    fork(); /*此处忽略错误检查  
  
*/  
    while (read(fd_in, buf, 2) > 0)  
    {  
        printf("%d: %s", getpid(), buf);  
        sprintf(buf, "%d Hello, World!\n", getpid());  
        write(fd_out, buf, strlen(buf));  
    }  
}
```

```
        sleep(1);
        memset(buf, 0, 1024);
    }
}
```

INFILE的内容是:

```
1
2
3
4
5
6
```

上面的程序中，父子进程都会去读INFILE，如果父子进程各维护各的文件偏移量，那么父子进程都会打印出1~6。

事实如何呢？请看输出内容：

```
manu@manu-hacks:~/code/self/c/fork$ ./fork_file
6602: 1
6603: 2
6602: 3
6603: 4
6602: 5
6603: 6
```

当然，有时候输出是这样的：

```
manu@manu-hacks:~/code/self/c/fork$ ./fork_file
6610: 1
6611: 2
6610: 3
6611: 4
6610: 5
6611: 5
6610: 6
```

如果父子进程各自维护自己的文件偏移量，那么一定是打印出两套1~6，但是事实并非如此。无论父进程还是子进程调用read函数导致文件偏移量后移都会被对方获知，这表明父子进程共用了一套文件偏移量。

对于第二个输出，为什么父子进程都打印5呢？这是因为我的机器是多核的，父子进程同时执行，发现当前文件偏移量是4*2，然后各自去读了第8和第9字节，也就是“5\n”。

写文件也是一样，如果fork之前打开了某文件，之后父子进程写入同一个文件描述符而又不采取任何同步的手段，那么就会因为共享文件偏移量而使输出相互混合，不可阅读。

文件描述符还有一个文件描述符标志（file descriptor flag）。目前只定义了一个标志位：FD_CLOSEEXEC，这是close_on_exec标志位。细心阅读open函数手册也会发现，open函数也有一个类似的标志位，即O_CLOSEEXEC，该标志位也是用于设置文件描述符标志的。

那么这个标志位到底有什么作用呢？如果文件描述符中将这个标志位置位，那么调用exec时会自动关闭对应的文件。

可是为什么需要这个标志位呢？主要是出于安全的考虑。

对于fork之后子进程执行exec这种场景，如果子进程可以操作父进程打开的文件，就会带来严重的安全隐患 [1]。一般来讲，调用exec的子进程时，因为它会另起炉灶，因此父进程打开的文件描述符也应该一并关闭，但事实上内核并没有主动这样做。试想如下场景，Webserver首先以root权限启动，打开只有拥有root权限才能打开的端口和日志等文件，再降到普通用户，fork出一些worker进程，在进程中进行解析脚本、写日志、输出结果等操作。由于子进程完全可以操作父进程打开的文件，因此子进程中的脚本只要继续操作这些文件描述符，就能越权操作root用户才能操作的文件。

为了解决这个问题，Linux引入了close on exec机制。设置了FD_CLOSEEXEC标志位的文件，在子进程调用exec家族函数时会将相应的文件关闭。而设置该标志位的方法有两种：

- open时，带上O_CLOSEEXEC标志位。
- open时如果未设置，那就在后面调用fcntl函数的F_SETFD操作来设置。

建议使用第一种方法。原因是第二种方法在某些时序条件下并不那么绝对的安全。考虑图4-7的场景：Thread 1还没来得及将FD_CLOSEEXEC置位，由于Thread 2已经执行过fork，这时候fork出来的子进程就不会关闭相应的文件。尽管Thread1后来调用了fcntl的F_SETFD操作，但是为时已晚，文件已经泄露了。

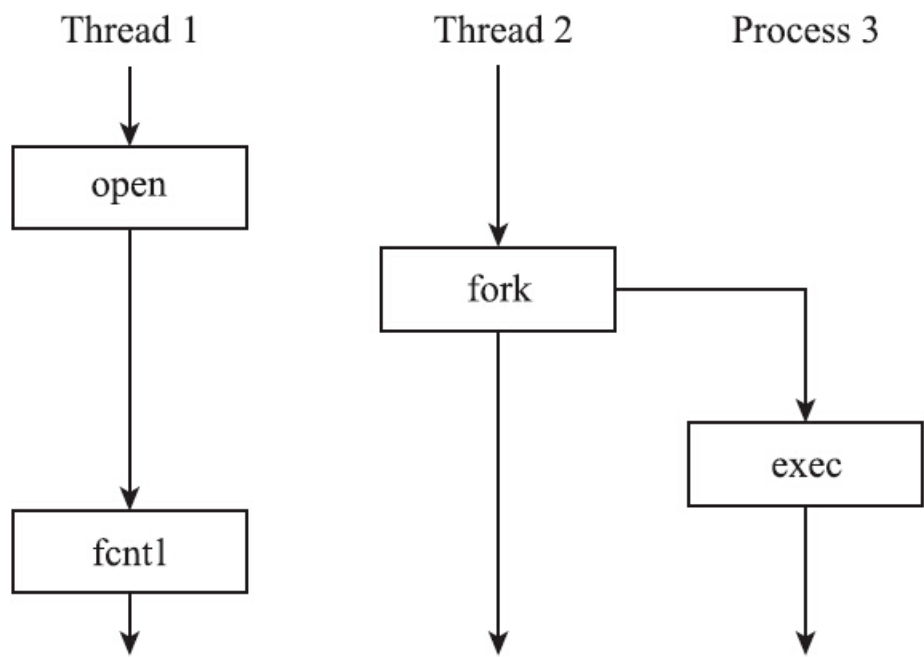


图4-7 未及时fcntl导致文件描述符的泄露



注意 图4-7中，多线程程序执行了fork，仅仅是为了示意，实际中并不鼓励这种做法。正相反，这种做法是十分危险的。多线程程序不应该调用fork来创建子进程，第8章会分析具体原因。

前面提到，执行fork时，子进程会获取父进程所有文件描述符的副本，但是测试结果表明，父子进程共享了文件的很多属性。这到底是怎么回事？让我们深入内核一探究竟。

[1] Linux系统文件描述符继承带来的危害请参看：<http://www.80sec.com/security-issue-on-linux-fd-inheritance.html>。

4.3.3 文件描述符复制的内核实现

在内核的进程描述符task_struct结构体中，与打开文件相关的变量如下所示：

```
struct task_struct {
    ...struct files_struct *files;...
}
```

调用fork时，内核会在copy_files函数中处理拷贝父进程打开的文件的相关事宜：

```
static int copy_files(unsigned long clone_flags,
                     struct task_struct *tsk)
{
    struct files_struct *oldf, *newf;
    int error = 0;
    oldf = current->files;
    if (!oldf)
        goto out;
    /*创建线程和
    vfork, 都不用复制父进程的文件描述符, 增加引用计数即可

    */
    if (clone_flags & CLONE_FILES) {
        atomic_inc(&oldf->count);
        goto out;
    }
    /*对于
    fork而言, 需要复制父进程的文件描述符

    */
    newf = dup_fd(oldf, &error);
    if (!newf)
        goto out;
    tsk->files = newf;
    error = 0;
out:
    return error;
}
```

CLONE_FILES标志位用来控制是否共享父进程的文件描述符。如果该标志位置位，则表示不必费劲复制一份父进程的文件描述符了，增加引用计数，直接共用一份就可以了。对于vfork函数和创建线程的pthread_create函数来说都是如此。但是fork函数却不同，调用fork函数时，该标志位为0，表示需要为子进程拷贝一份父进程的文件描述符。文件描述符的拷贝是通过内核的dup_fd函数来完成的。

```
struct files_struct *dup_fd(struct files_struct *oldf,
                           int *errorp)
{
    struct file_struct *newf;
    struct file *old_fds, **new_fds;
    int open_files, size, i;
    struct fdtable *old_fdt, *new_fdt;
    *errorp = -ENOMEM;
    newf = kmem_cache_alloc(files_cachep, GFP_KERNEL);
    if (!newf)
        goto out;
```

dup_fd函数首先会给予进程分配一个file_struct结构体，然后做一些赋值操作。这个结构体是进程描述符中与打开文件相关的数据结构，每一个打开的文件都会记录在该结构体中。其定义代码如下：

```
struct files_struct {
    atomic_t count;
    struct fdtable __rcu *fdt;
    struct fdtable fdtab;
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    int next_fd;
    struct embedded_fd_set close_on_exec_init;
    struct embedded_fd_set open_fds_init;
    struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
struct fdtable
{
    unsigned int max_fds;
    struct file __rcu **fd; /* current fd array */
    fd_set *close_on_exec;
    fd_set *open_fds;
    struct rcu_head rcu;
    struct fdtable *next;
};
struct embedded_fd_set {
    unsigned long fds_bits[1];
};
```

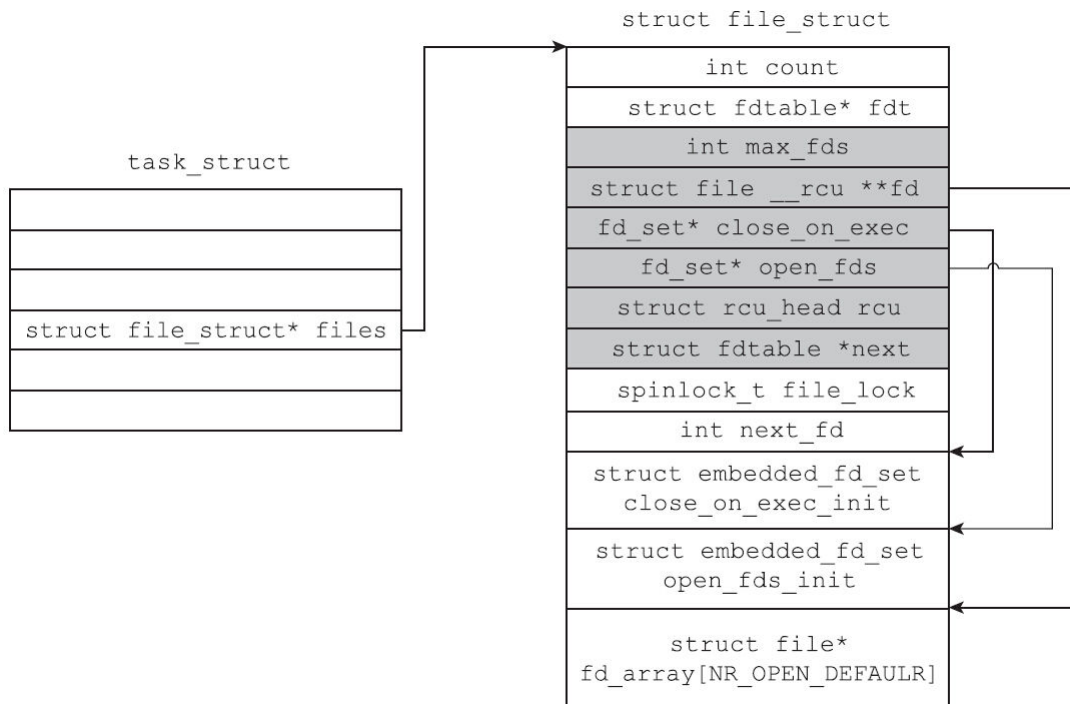
初看之下struct fdtable的内容与struct files_struct的内容有颇多重复之处，包括close_on_exec文件描述符位图、打开文件描述符位图及file指针数组等，但事实上并非如此。struct files_struct中的成员是相应数据结构的实例，而struct fdtable中的成员是相应的指针。

Linux系统假设大多数的进程打开的文件不会太多。于是Linux选择了一个long类型的位数（32位系统下为32位，64位系统下为64位）作为经验值。

以64位系统为例，file_struct结构体自带了可以容纳64个struct file类型指针的数组fd_array，也自带了两个大小为64的位图，其中open_fds_init位图用于记录文件的打开情况，close_on_exec_init位图用于记录文件描述符的FD_CLOSEXCE标志位是否置位。只要进程打开的文件个数小于64，file_struct结构体自带的指针数组和两个位图就足以满足需要。因此在分配了file_struct结构体后，内核会初始化file_struct自带的fdtable，代码如下所示：

```
atomic_set(&newf->count, 1);
spin_lock_init(&newf->file_lock);
newf->next_fd = 0;
```


初始化之后，子进程的file_struct的情况如图4-8所示。注意，此时file_struct结构体中的fdt指针并未指向file_struct自带的struct fdtable类型的fdtab变量。原因很简单，因为此时内核还没有检查父进程打开文件的个数，因此并不确定自带的结构体能否满足需要。



接下来，内核会检查父进程打开文件的个数。如果父进程打开的文件超过了64个，`struct files_struct`中自带的数组和位图就不能满足需要了。这种情况下内核会分配一个新的`struct fdtable`，代码如下：

`alloc_fdtable`所做的事情，不过是分配`fdtable`结构体本身，以及分配一个指针数组和两个位图（如图4-9所示）。分配之前会根据父进程打开文件的数目，计算出一个合理的值`nr`，以确保分配的数组和位图能够满足需要。

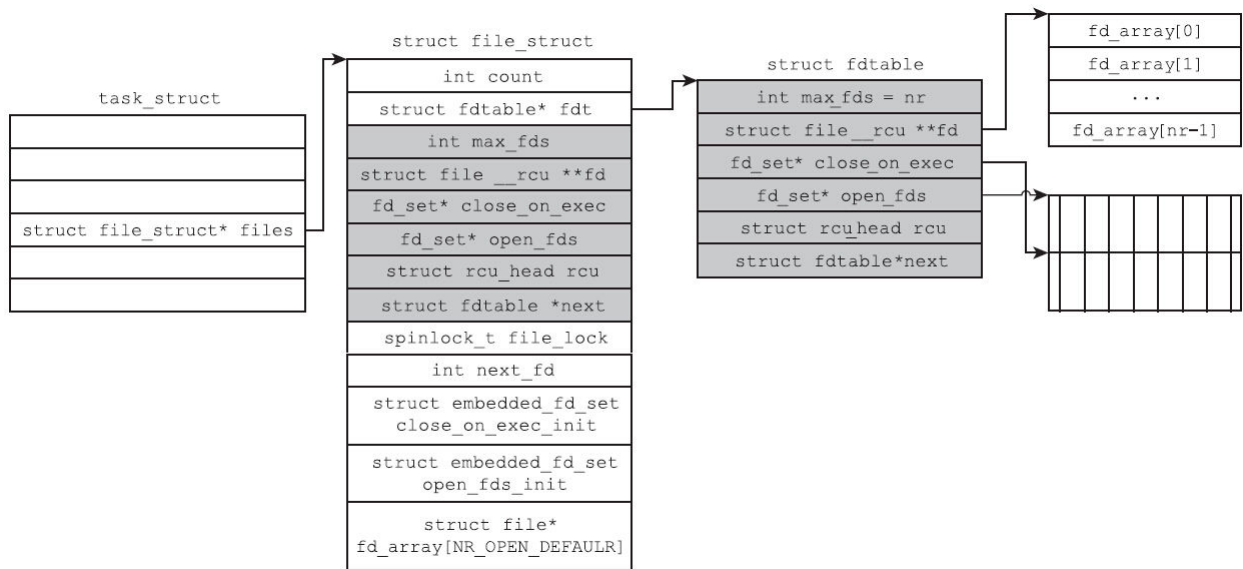


图4-9 alloc_fdtbl原理

无论是使用file_struct结构体自带的fdtable，还是使用alloc_fdtbl分配的fdtable，接下来要做的事情都一样，即将父进程的两个位图信息和打开文件的struct file类型指针拷贝到子进程的对应数据结构中，代码如下：

```
old_fds = old_fdt->fd; /*父进程的

struct file 指针数组

*/new_fds = new_fdt->fd; /*子进程的

struct file 指针数组

*//* 拷贝打开文件位图

*/memcpy(new_fdt->open_fds->fds_bits,old_fdt->open_fds->fds_bits, open_files/8);/* 拷贝

close_on_exec位图

*/memcpy(new_fdt->close_on_exec->fds_bits,old_fdt->close_on_exec->fds_bits, open_files/8);for (i = open_files; i != 0; i--) { struct file *f = *old_fds++; if (f) {

1 */ } else { FD_CLR(open_files - i, new_fdt->open_fds); }/* 子进程的

struct file类型指针.

*指向和父进程相同的

struct file 结构体

*/ rcu_assign_pointer(*new_fds++, f); }spin_unlock(&oldf->file_lock);/* compute the remainder to be cleared */size = (new_fdt->max_fds - open_files) * sizeof(struct file *

struct file结构的指针清零

*/memset(new_fds, 0, size);/*将尚未分配到的位图区域清零

*/if (new_fdt->max_fds > open_files) { int left = (new_fdt->max_fds-open_files)/8; int start = open_files / (8 * sizeof(unsigned long)); memset(&new_fdt->open_fds->fds
}
```



注意 procfs的/proc/PID/status中的FDSize，记录了当前fdtable的大小：

```
manu@manu-hacks:~$ cat /proc/1/status
FDSize: 128
```

当然了，FDSize记录的是目前fdtable能容纳的struct file指针，而不是已经打开的文件个数，已经打开的文件记录在/proc/PID/fd中。

通过对上述流程的梳理，不难看出，父子进程之间拷贝的是struct file的指针，而不是struct file的实例，父子进程的struct file类型指针，都指向同一个struct file实例。fork之后，父子进程的文件描述符关系如图4-10所示。

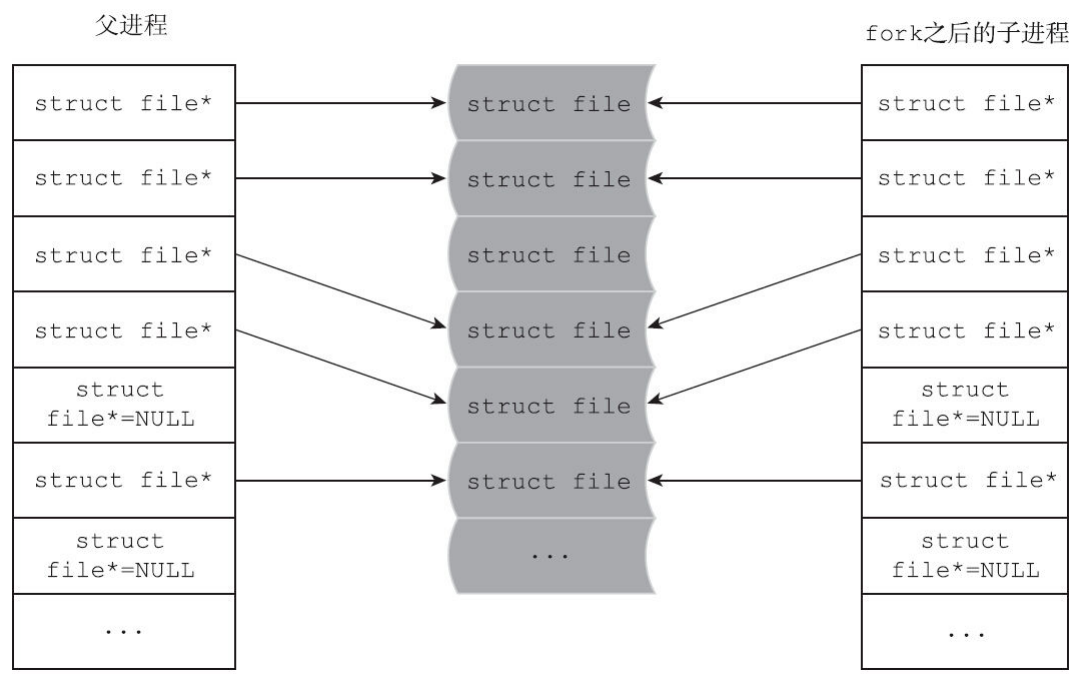


图4-10 fork之后，父子进程的文件描述符关系

下面来看看struct file成员变量：

```
struct file{
    ...
    unsigned int  f_flags
    fmode_t      f_mode
    loff_t        f_pos; /*文件位置指针的当前值，即文件偏移量

*/
    ...
}
```

看到此处，就不难理解父子进程是如何共享文件偏移量的了，那是因为父子进程的指针都指向了同一个struct file结构体。

4.4 进程的创建之vfork（）

在早期的实现中，fork没有实现写时拷贝机制，而是直接对父进程的数据段、堆和栈进行完全拷贝，效率十分低下。很多程序在fork一个子进程后，会紧接着执行exec家族函数，这更是一种浪费。所以BSD引入了vfork。既然fork之后会执行exec函数，拷贝父进程的内存数据就变成了一种无意义的行为，所以引入的vfork压根就不会拷贝父进程的内存数据，而是直接共享。再后来Linux引入了写时拷贝的机制，其效率提高了很多，这样一来，vfork其实就可以退出历史舞台了。除了一些需要将性能优化到极致的场景，大部分情况下不需要再使用vfork函数了。

vfork会创建一个子进程，该子进程会共享父进程的内存数据，而且系统将保证子进程先于父进程获得调度。子进程也会共享父进程的地址空间，而父进程将被一直挂起，直到子进程退出或执行exec。

注意，vfork之后，子进程如果返回，则不要调用return，而应该使用_exit函数。如果使用return，就会出现诡异的错误^[1]。请看下面的示例代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int glob = 88 ;
int main(void) {
    int var;
    var = 88;
    pid_t pid;
    if ((pid = vfork()) < 0) {
        printf("vfork error");
        exit(-1);
    } else if (pid == 0) { /* 子进程

        */
        var++;
        glob++;
        return 0;
    } printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
    return 0;
}
```

调用子进程，如果使用return返回，就意味着main函数返回了，因为栈是父子进程共享的，所以程序的函数栈发生了变化。main函数return之后，通常会调用exit系的函数，父进程收到子进程的exit之后，就会开始从vfork返回，但是这时整个main函数的栈都已经不复存在了，所以父进程压根无法执行。于是会返回一个诡异的栈地址，对于在某些内核版本中，进程会直接报栈错误然后退出，但是在某些内核版本中，有可能就会再次进出main，于是进入一个无限循环，直到vfork返回错误。笔者的Ubuntu版本就是后者。

一般来说，vfork创建的子进程会执行exec，执行完exec后应该调用_exit返回。注意是_exit而不是exit。因为exit会导致父进程stdio缓冲区的冲刷和关闭。我们会在后面讲述exit和_exit的区别。

^[1] 请参考著名程序员陈皓的《vfork挂掉的一个问题》一文。

4.5 daemon进程的创建

daemon进程又被称为守护进程，一般来说它有以下两个特点：

- 生命周期很长，一旦启动，正常情况下不会终止，一直运行到系统退出。但凡事无绝对：daemon进程其实也是可以停止的，如很多daemon提供了stop命令，执行stop命令就可以终止daemon，或者通过发送信号将其杀死，又或者因为daemon进程代码存在bug而异常退出。这些退出一般都是由手工操作或因异常引发的。

- 在后台执行，并且不与任何控制终端相关联。即使daemon进程是从终端命令行启动的，终端相关的信号如SIGINT、SIGQUIT和SIGTSTP，以及关闭终端，都不会影响到daemon进程的继续执行。

习惯上daemon进程的名字通常以d结尾，如sshd、rsyslogd等。但这仅仅是习惯，并非一定要如此。

如何使一个进程变成daemon进程，或者说编写daemon进程，需要遵循哪些规则或步骤呢？

一般来讲，创建一个daemon进程的步骤被概括地称为double-fork magic。细细说来，需要以下步骤。

(1) 执行fork()函数，父进程退出，子进程继续

执行这一步，原因有二：

- 父进程有可能是进程组的组长（在命令行启动的情况下），从而不能够执行后面要执行的setsid函数，子进程继承了父进程的进程组ID，并且拥有自己的进程ID，一定不会是进程组的组长，所以子进程一定可以执行后面要执行的setsid函数。

- 如果daemon是从终端命令行启动的，那么父进程退出会被shell检测到，shell会显示shell提示符，让子进程在后台执行。

(2) 子进程执行如下三个步骤，以摆脱与环境的关系

1) 修改进程的当前目录为根目录 (/)。

这样做是有原因的，因为daemon一直在运行，如果当前工作路径上包含有根文件系统以外的其他文件系统，那么这些文件系统将无法卸载。因此，常规是将当前工作目录切换成根目录，当然也可以是其他目录，只要确保该目录所在的文件系统不会被卸载即可。

2) 调用setsid函数。这个函数的目的是切断与控制终端的所有关系，并且创建一个新的会话。

这一步比较关键，因为这一步确保了子进程不再归属于控制终端所关联的会话。因此无论终端是否发送SIGINT、SIGQUIT或SIGTSTP信号，也无论终端是否断开，都与要创建的daemon进程无关，不会影响到daemon进程的继续执行。

3) 设置文件模式创建掩码为0。

```
umask(0)
```

这一步的目的是让daemon进程创建文件的权限属性与shell脱离关系。因为默认情况下，进程的umask来源于父进程shell的umask。如果不执行umask(0)，那么父进程shell的umask就会影响到daemon进程的umask。如果用户改变了shell的umask，那么也就相当于改变了daemon的umask，就会造成daemon进程每次执行的umask信息可能会不一致。

(3) 再次执行fork，父进程退出，子进程继续

执行完前面两步之后，可以说已经比较圆满了：新建会话，进程是会话的首进程，也是进程组的首进程。进程ID、进程组ID和会话ID，三者的值相同，进程和终端无关联。那么这里为何还要再执行一次fork函数呢？

原因是，daemon进程有可能会打开一个终端设备，即daemon进程可能会根据需要，执行类似如下代码：

```
int fd = open("/dev/console", O_RDWR);
```

这个打开的终端设备是否会成为daemon进程的控制终端，取决于两点：

- daemon进程是不是会话的首进程。

- 系统实现。（BSD风格的实现不会成为daemon进程的控制终端，但是POSIX标准说这由具体实现来决定）。

既然如此，为了确保万无一失，只有确保daemon进程不是会话的首进程，才能保证打开的终端设备不会自动成为控制终端。因此，不得不执行第二次fork，fork之后，父进程退出，子进程继续。这时，子进程不再是会话的首进程，也不是进程组的首进程了。

(4) 关闭标准输入（stdin）、标准输出（stdout）和标准错误（stderr）

因为文件描述符0、1和2指向的就是控制终端。daemon进程已经不再与任意控制终端相关联，因此这三者都没有意义。一般来讲，关闭了之后，会打开/dev/null，并执行dup2函数，将0、1和2重定向到/dev/null。这个重定向是有意义的，防止了后面的程序在文件描述符0、1和2上执行I/O库函数而导致报错。

至此，即完成了daemon进程的创建，进程可以开始自己真正的工作了。

上述步骤比较繁琐，对于C语言而言，glibc提供了daemon函数，从而帮我们将程序转化成daemon进程。

```
#include <unistd.h>
int daemon(int nochdir, int noclose);
```

该函数有两个入参，分别控制一种行为，具体如下。

其中的nochdir，用来控制是否将当前工作目录切换到根目录。

·0：将当前工作目录切换到/。

·1：保持当前工作目录不变。

而noclose，用来控制是否将标准输入、标准输出和标准错误重定向到/dev/null。

·0：将标准输入、标准输出和标准错误重定向到/dev/null。

·1：保持标准输入、标准输出和标准错误不变。

一般情况下，这两个入参都要为0。

```
ret = daemon(0,0)
```

成功时，daemon函数返回0；失败时，返回-1，并置errno。因为daemon函数内部会调用fork函数和setsid函数，所以出错时errno可以查看fork函数和setsid函数的出错情形。

glibc的daemon函数做的事情，和前面讨论的大体一致，但是做得并不彻底，没有执行第二次的fork。

4.6 进程的终止

在不考虑线程的情况下，进程的退出有以下5种方式。

正常退出有3种：

- 从main函数return返回

- 调用exit

- 调用_exit

异常退出有两种：

- 调用abort

- 接收到信号，由信号终止

4.6.1 _exit函数

_exit函数的接口定义如下：

```
#include <unistd.h>
void _exit(int status);
```

_exit函数中status参数定义了进程的终止状态，父进程可以通过wait（）来获取该状态值。

需要注意的是返回值，虽然status是int型，但是仅有低8位可以被父进程所用。所以写exit（-1）结束进程时，在终端执行“\$？”会发现返回值是255。

如果是shell相关的编程，shell可能需要获取进程的退出值，那么退出值最好不要大于128。如果退出值大于128，会给shell带来困扰。POSIX标准规定了退出状态及其含义如表4-2所示。

表4-2 shell编程中退出状态及其含义

值	含 义
0	命令成功执行并退出
1~125	命令未成功地退出，具体含义由各自的命令来定义
126	命令找到了，文件无法执行

(续)

值	含 义
127	命令找不到
>128	命令因收到信号而死亡

下面的命令被SIGINT信号（signo=2）中断，返回了130。如程序通过exit返回130，与其配合工作的shell就可能会误判为收到信号而退出。

```
manu@manu-hacks:~/code/me/exit$ sleep 10000
^C
manu@manu-hacks:~/code/me/exit$ $?
130: 未找到命令
```

用户调用_exit函数，本质上是调用exit_group系统调用。这点在前面已经详细介绍过，在此就不再赘述了。

4.6.2 exit函数

exit函数更常见一些，其接口定义如下：

```
#include <stdlib.h>
void exit(int status);
```

exit（）函数的最后也会调用_exit（）函数，但是exit在调用_exit之前，还做了其他工作：

- 1）执行用户通过调用atexit函数或on_exit定义的清理函数。
- 2）关闭所有打开的流（stream），所有缓冲的数据均被写入（flush），通过tmpfile创建的临时文件都会被删除。
- 3）调用_exit。

图4-11给出了exit函数和_exit函数的差异。

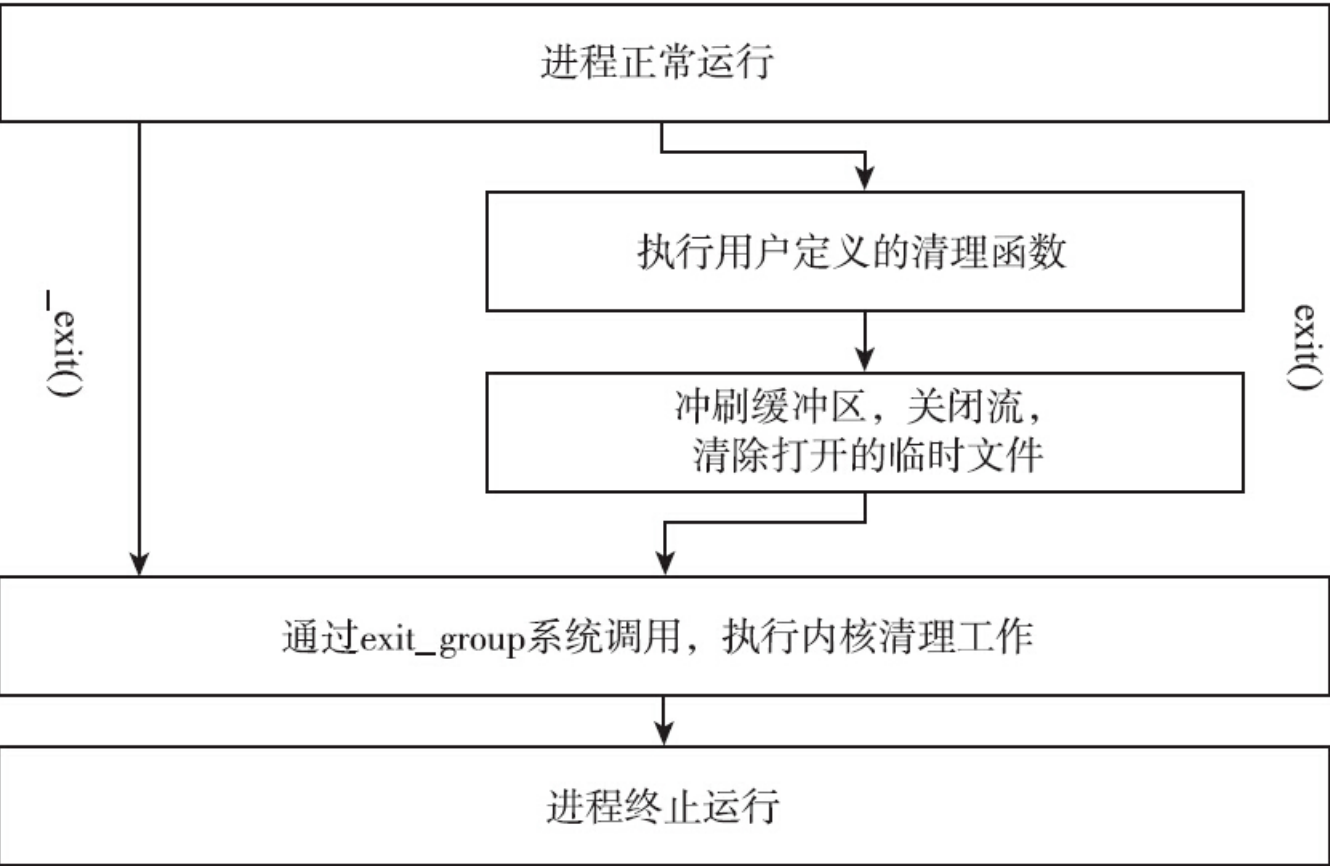


图4-11 exit和_exit比较

下面介绍exit函数和_exit函数的不同之处。

首先是exit函数会执行用户注册的清理函数。用户可以通过调用atexit（）函数或on_exit（）函数来

定义清理函数。这些清理函数在调用**return**或调用**exit**时会被执行。执行顺序与函数注册的顺序相反。当进程收到致命信号而退出时，注册的清理函数不会被执行；当进程调用**_exit**退出时，注册的清理函数不会被执行；当执行到某个清理函数时，若收到致命信号或清理函数调用了**_exit()**函数，那么该清理函数不会返回，从而导致排在后面的需要执行的清理函数都会被丢弃。

其次是**exit**函数会冲刷（flush）标准I/O库的缓冲并关闭流。**glibc**提供的很多与I/O相关的函数都提供了缓冲区，用于缓存大块数据。

缓冲有三种方式：无缓冲（**_IONBF**）、行缓冲（**_IOLBF**）和全缓冲（**_IOFBF**）。

·无缓冲：就是没有缓冲区，每次调用**stdio**库函数都会立刻调用**read/write**系统调用。

·行缓冲：对于输出流，收到换行符之前，一律缓冲数据，除非缓冲区满了。对于输入流，每次读取一行数据。

·全缓冲：就是缓冲区满之前，不会调用**read/write**系统调用来进行读写操作。

对于后两种缓冲，可能会出现这种情况：进程退出时，缓冲区里面可能还有未冲刷的数据。如果不冲刷缓冲区，缓冲区的数据就会丢失。比如行缓冲迟迟没有等到换行符，又或者全缓冲没有等到缓冲区满。尤其是后者，很容易出现，因为**glibc**的缓冲区默认是8192字节。**exit**函数在关闭流之前，会冲刷缓冲区的数据，确保缓冲区里的数据不会丢失。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void foo()
{
    fprintf(stderr,"foo says bye.\n");
}
void bar()
{
    fprintf(stderr,"bar says bye.\n");
}
int main(int argc, char **argv)
{
    atexit(foo);
    atexit(bar);
    fprintf(stdout,"Oops ... forgot a newline!");
    sleep(2);
    if (argc > 1 && strcmp(argv[1],"exit") == 0)
        exit(0);
    if (argc > 1 && strcmp(argv[1],"_exit") == 0)
        _exit(0);
    return 0;
}
```

注意上面的示例代码，**fprintf**打印的字符串是没有换行符的，对于标准输出流**stdout**，采用的是行缓冲，收到换行符之前是不会有输出的。输出情况如下：

```
manu@manu-hacks:exit$ ./test exit
bar says bye.
foo says bye.
Oops ... forgot a newline!manu@manu-hacks:exit$
manu@manu-hacks:exit$
manu@manu-hacks:exit$ ./test
```

```
bar says bye.  
foo says bye.  
Oops ... forgot a newline!manu@manu-hacks:exit$  
manu@manu-hacks:exit$  
manu@manu-hacks:exit$ ./test _exit  
manu@manu-hacks:~/code/self/c/exit$
```

尽管缓冲区里的数据没有等到换行符，但是无论是调用`return`返回还是调用`exit`返回，缓冲区里的数据都会被冲刷，“Oops...forgot a newline!”都会被输出。因为`exit()`函数会负责此事。从测试代码的输出也可以看出，`exit()`函数首先执行的是用户注册的清理函数，然后才执行了缓冲区的冲刷。

第三，存在临时文件，`exit`函数会负责将临时文件删除，这点在第3章中已经介绍过，此处就不再赘述了。

`exit`函数的最后调用了`_exit()`函数，最终殊途同归，走向内核清理。

4.6.3 return退出

`return`是一种更常见的终止进程的方法。执行`return (n)`等同于执行`exit (n)`，因为调用`main ()`的运行时函数会将`main`的返回值当作`exit`的参数。

4.7 等待子进程

4.7.1 僵尸进程

进程就像一个生命体，通过`fork()`函数，子进程呱呱坠地。有的子进程子承父业，继续执行与父进程一样的程序（相同的代码段，尽管可能是不同的程序分支），有的子进程则比较叛逆，通过`exec`离家出走，走向与父进程完全不同的道路。

令人悲伤的是，如同所有的生命体一样，进程也会消亡。进程退出时会进行内核清理，基本就是释放进程所有的资源，这些资源包括内存资源、文件资源、信号量资源、共享内存资源，或者引用计数减一，或者彻底释放。不过，进程的退出其实并没有将所有的资源完全释放，仍保留了少量的资源，比如进程的PID依然被占用着，不可被系统分配。此时的进程不可运行，事实上也没有地址空间让其运行，进程进入僵尸状态。

为什么进程退出之后不将所有的资源释放，从此灰飞烟灭，一了百了，反而非要保留少量资源，进入僵尸状态呢？看看僵尸进程依然占有的系统资源，我们就能获得答案。僵尸进程依然保留的资源有进程控制块`task_struct`、内核栈等。这些资源不释放是为了提供一些重要的信息，比如进程为何退出，是收到信号退出还是正常退出，进程退出码是多少，进程一共消耗了多少系统CPU时间，多少用户CPU时间，收到了多少信号，发生了多少次上下文切换，最大内存驻留集是多少，产生多少缺页中断？等等。这些信息，就像墓志铭，总结了进程的一生。如果没有这个僵尸状态，进程的这些信息也会随之流逝，系统也将再也没有机会获知该进程的相关信息了。因此进程退出后，会保留少量的资源，等待父进程前来收集这些信息。一旦父进程收集了这些信息之后（通过调用下面提到的`wait/waitpid`等函数），这些残存的资源完成了它的使命，就可以释放了，进程就脱离僵尸状态，彻底消失了。

从上面的讨论可以看出，制造一个僵尸进程是一件很容易的事情，只要父进程调用`fork`创建子进程，子进程退出后，父进程如果不调用`wait`或`waitpid`来获取子进程的退出信息，子进程就会沦为僵尸进程。示例代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    pid=fork();
    if(pid<0)
    {
        /* 如果出错
```

```

*/
    printf("error occurred!\n");
}
else if(pid==0)
{
    /* 子进程

*/
    exit(0);
}
else
{
    /* 父进程

*/
    sleep(300); /* 休眠

300秒

*/
    wait(NULL); /* 获取僵尸进程的退出信息

*/
}
return 0;
}

```

上面的例子中父进程休眠300秒后才会调用wait来获取子进程的退出信息。而子进程退出之后会变成僵尸状态，苦苦等待父进程来获取退出信息。在这300秒左右的时间里，子进程就是一个僵尸进程。

如何查看一个进程是否处于僵尸状态呢？ps命令输出的进程状态Z，就表示进程处于僵尸状态，另外procfs提供的status信息中的State给出的值是Z（zombie），也表明进程处于僵尸状态。

```

ps ax
.....
3940 pts/10  S      0:00 ./zombie
3941 pts/10  Z      0:00 [zombie] <defunct>
cat /proc/3941/status
Name:    zombie
State:   Z (zombie)
Tgid:    3941
Ngid:    0
Pid:     3941
PPid:    3940
.....

```

进程一旦进入僵尸状态，就进入了一种刀枪不入的状态，“杀人不眨眼”的kill-9也无能为力，因为谁也没有办法杀死一个已经死去的进程。

清除僵尸进程有以下两种方法：

- 父进程调用wait函数，为子进程“收尸”。
- 父进程退出，init进程会为子进程“收尸”。

一般而言，系统不希望大量进程长期处于僵尸状态，因为会浪费系统资源。除了少量的内存资源外，比较重要的是进程ID。僵尸进程并没有将自己的进程ID归还给系统，而是依然占有这个进程ID，因此系统不能将该ID分配给其他进程。

对于编程来说，如何防范僵尸进程的产生呢？答案是具体情况具体分析。

如果我们不关心子进程的退出状态，就应该将父进程对SIGCHLD的处理函数设置为SIG_IGN，或者在调用sigaction函数时设置SA_NOCLDWAIT标志位。这两者都会明确告诉子进程，父进程很“绝情”，不会为子进程“收尸”。子进程退出的时候，内核会检查父进程的SIGCHLD信号处理结构体是否设置了SA_NOCLDWAIT标志位，或者是否将信号处理函数显式地设为SIG_IGN。如果是，则autoreap为true，子进程发现autoreap为true也就“死心”了，不会进入僵尸状态，而是调用release_task函数“自行了断”了。

如果父进程关心子进程的退出信息，则应该在流程上妥善设计，能够及时地调用wait，使子进程处于僵尸状态的时间不会太久。

对于创建了很多子进程的应用来说，知道子进程的返回值是有意义的。比如说父进程维护一个进程池，通过进程池里的子进程来提供服务。当子进程退出的时候，父进程需要了解子进程的返回值来确定子进程的“死因”，从而采取更有针对性的措施。

4.7.2 等待子进程之wait（）

Linux提供了wait（）函数来获取子进程的退出状态：

```
include <sys/wait.h>
pid_t wait(int *status);
```

成功时，返回已退出子进程的进程ID；失败时，则返回-1并设置errno，常见的errno及说明见表4-3。

表4-3 wait函数的出错情况

errno	说 明
ECHLD	调用进程时发现并没有子进程需要等待
EINTR	函数被信号中断

注意父子进程是两个进程，子进程退出和父进程调用wait（）函数来获取子进程的退出状态在时间上是独立的事件，因此会出现以下两种情况：

- 子进程先退出，父进程后调用wait（）函数。
- 父进程先调用wait（）函数，子进程后退出。

对于第一种情况，子进程几乎已经销毁了自己所有的资源，只留下少量的信息，苦苦等待父进程来“收尸”。当父进程调用wait（）函数的时候，苦守寒窑十八载的子进程终于等到了父进程来“收尸”，这种情况下，父进程获取到子进程的状态信息，wait函数立刻返回。

对于第二种情况，父进程先调用wait（）函数，调用时并无子进程退出，该函数调用就会陷入阻塞状态，直到某个子进程退出。

wait（）函数等待的是任意一个子进程，任何一个子进程退出，都可以让其返回。当多个子进程都处于僵尸状态，wait（）函数获取到其中一个子进程的信息后立刻返回。由于wait（）函数不会接受pid_t类型的入参，所以它无法明确地等待特定的子进程。

一个进程如何等待所有的子进程退出呢？wait（）函数返回有三种可能性：

- 等到了子进程退出，获取其退出信息，返回子进程的进程ID。
- 等待过程中，收到了信号，信号打断了系统调用，并且注册信号处理函数时并没有设置SA_RESTART标志位，系统调用不会被重启，wait（）函数返回-1，并且将errno设置为EINTR。
- 已经成功地等待了所有子进程，没有子进程的退出信息需要接收，在这种情况下，wait（）函数返回-1，errno为ECHILD。

《Linux/Unix系统编程手册》给出下面的代码来等待所有子进程的退出：

```
while((childPid = wait(NULL)) != -1)
    continue;
if(errno !=ECHILD)
    errExit("wait");
```

这种方法并不完全，因为这里忽略了wait（）函数被信号中断这种情况，如果wait（）函数被信号中断，上面的代码并不能成功地等待所有子进程退出。

若将上面的wait（）函数封装一下，使其在信号中断后，自动重启wait就完备了。代码如下：

```
pid_t r_wait(int *stat_loc)
{
    int retval;
    while(((retval = wait(stat_loc)) == -1 &&
           (errno == EINTR))
           ;
    return retval;
}
while((childPid = r_wait(NULL)) != -1)
    continue;
If(errno != ECHILD)
{
    /*some error happened*/
}
```

如果父进程调用wait（）函数时，已经有多个子进程退出且都处于僵尸状态，那么哪一个子进程会被先处理是不一定的（标准并未规定处理的顺序）。

通过上面的讨论，可以看出wait（）函数存在一定的局限性：

- 不能等待特定的子进程。如果进程存在多个子进程，而它只想获取某个子进程的退出状态，并不关心其他子进程的退出状态，此时wait（）只能一一等待，通过查看返回值来判断是否为关心的子进程。

- 如果不存在子进程退出，wait（）只能阻塞。有些时候，仅仅是想尝试获取退出子进程的退出状态，如果不存在子进程退出就立刻返回，不需要阻塞等待，类似于trywait的概念。wait（）函数没有提供trywait的接口。

- wait（）函数只能发现子进程的终止事件，如果子进程因某信号而停止，或者停止的子进程收到SIGCONT信号又恢复执行，这些事件wait（）函数是无法获知的。换言之，wait（）能够探知子进程的死亡，却不能探知子进程的昏迷（暂停），也无法探知子进程从昏迷中苏醒（恢复执行）。

由于上述三个缺点的存在，所以Linux又引入了waitpid（）函数。

4.7.3 等待子进程之waitpid ()

waitpid () 函数接口如下：

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

先说说waitpid () 与wait () 函数相同的地方：

- 返回值的含义相同，都是终止子进程或因信号停止或因信号恢复而执行的子进程的进程ID。
- status的含义相同，都是用来记录子进程的相关事件，后面一节将会详细介绍。

接下来介绍waitpid () 函数特有的功能。

其第一个参数是pid_t类型，有了此值，不难看出waitpid函数肯定具备了精确打击的能力。waitpid函数可以明确指定要等待哪一个子进程的退出（以及停止和恢复执行）。事实上，扩展的功能不仅仅如此：

- pid>0：表示等待进程ID为pid的子进程，也就是上文提到的精确打击的对象。
- pid=0：表示等待与调用进程同一个进程组的任意子进程；因为子进程可以设置自己的进程组，所以某些子进程不一定和父进程归属于同一个进程组，这样的子进程，waitpid函数就毫不关心了。
- pid=-1：表示等待任意子进程，同wait类似。waitpid (-1, &status, 0) 与wait (&status) 完全等价。
- pid<-1：等待所有子进程中，进程组ID与pid绝对值相等的所有子进程。

内核之中，wait函数和waitpid函数调用的都是wait4系统调用。下面是wait4系统调用的实现。函数的中间部分，根据pid的正负或是否为0和-1来定义wait_opts类型的变量wo，后面会根据wo来控制到底关心哪些进程的事件。

```
SYSCALL_DEFINE4(wait4, pid_t, upid, int __user *, stat_addr,
                 int, options, struct rusage __user *, ru)
{
    struct wait_opts wo;
    struct pid *pid = NULL;
    enum pid_type type;
    long ret;
    if (options & ~(WNOHANG|WUNTRACED|WCONTINUED|
                    __WNOTHREAD|__WCLONE|__WALL))
        return -EINVAL;
    if (upid == -1)
        type = PIDTYPE_MAX;    /*任意子进程
```

```

*/
else if (upid < 0) {
    type = PIDTYPE_PGID;
    pid = find_get_pid(-upid);
} else if (upid == 0) {
    type = PIDTYPE_PGID;
    pid = get_task_pid(current, PIDTYPE_PGID);
} else /* upid > 0 */ {
    type = PIDTYPE_PID;
    pid = find_get_pid(upid);
}
wo.wo_type = type;
wo.wo_pid = pid;
wo.wo_flags = options | WEXITED;
wo.wo_info = NULL;
wo.wo_stat = stat_addr;
wo.wo_rusage = ru;
ret = do_wait(&wo);
put_pid(pid);
/* avoid REGPARM breakage on x86: */
asmlinkage_protect(4, ret, upid, stat_addr, options, ru);
return ret;
}

```

可以看到，内核的do_wait函数会根据wait_opts类型的wo变量来控制到底在等待哪些子进程的状态。

当前进程中的每一个线程（在内核层面，线程就是进程，每个线程都有独立的task_struct），都会遍历其子进程。在内核中，task_struct中的children成员变量是个链表头，该进程的所有子进程都会链入该链表，遍历起来比较方便。代码如下：

```

static int do_wait_thread(struct wait_opts *wo, struct task_struct *tsk)
{
    struct task_struct *p;
    list_for_each_entry(p, &tsk->children, sibling) {
        /*遍历进程所有的子进程

*/
        int ret = wait_consider_task(wo, 0, p);
        if (ret)
            return ret;
    }
    return 0;
}

```

但是我们并不一定关心所有的子进程。当wait（）函数或waitpid（）函数的第一个参数pid等于-1的时候，表示任意子进程我们都关心。但是如果是waitpid（）函数的其他情况，则表示我们只关心其中的某些子进程或某个子进程。内核需要对所有的子进程进行过滤，找到关心的子进程。这个过滤的环节是在内核的eligible_pid函数中完成的。

```

/* 当

```

```

waitpid的第一个参数为

```

```

-1时，

```

```

wo->wo_type 赋值为

```

```
PIDTYPE_MAX
* 其他三种情况
```

```
task_pid_type(p, wo->wo_type) == wo->wo_pid检验
```

```
* 或者检查
```

```
pid是否相等, 或者检查进程组
```

```
ID是否等于指定值
```

```
*/
static int eligible_pid(struct wait_opts *wo, struct task_struct *p)
{
    return wo->wo_type == PIDTYPE_MAX ||
           task_pid_type(p, wo->wo_type) == wo->wo_pid;
}
```

`waitpid`函数的第三个参数`options`是一个位掩码（bit mask），可以同时存在多个标志。当`options`没有设置任何标志位时，其行为与`wait`类似，即阻塞等待与`pid`匹配的子进程退出。

`options`的标志位可以是如下标志位的组合：

- WUNTRACE**：除了关心终止子进程的信息，也关心那些因信号而停止的子进程信息。

- WCONTINUED**：除了关心终止子进程的信息，也关心那些因收到信号而恢复执行的子进程的状态信息。

- WNOHANG**：指定的子进程并未发生状态变化，立刻返回，不会阻塞。这种情况下返回值是0。如果调用进程并没有与`pid`匹配的子进程，则返回-1，并设置`errno`为ECHILD，根据返回值和`errno`可以区分这两种情况。

传统的`wait`函数只关注子进程的终止，而`waitpid`函数则可以通过前两个标志位来检测子进程的停止和从停止中恢复这两个事件。

讲到这里，需要解释一下什么是“使进程停止”，什么是“使进程继续”，以及为什么需要这些。设想如下的场景，正在某机器上编译一个大型项目，编译过程需要消耗很多CPU资源和磁盘I/O资源，并且耗时很久。如果我暂时需要用机器做其他事情，虽然可能只需要占用几分钟时间。但这会使这几分钟内的用户体验非常糟糕，那怎么办？当然，杀掉编译进程是一个选择，但是这个方案并不好。因为编译耗时很久，贸然杀死进程，你将不得不从头编译起。这时候，我们需要的仅仅是让编译大型工程的

进程停下来，把CPU资源和I/O资源让给我，让我从容地做自己想做的事情，几分钟后，我用完了，让编译的进程继续工作就行了。

Linux提供了SIGSTOP（信号值19）和SIGCONT（信号值18）两个信号，来完成暂停和恢复的动作，可以通过执行kill-SIGSTOP或kill-19来暂停一个进程的执行，通过执行kill-SIGCONT或kill-18来让一个暂停的进程恢复执行。

waitpid（）函数可以通过WUNTRACE标志位关注停止的事件，如果有子进程收到信号处于暂停状态，waitpid就可以返回。

同样的道理，通过WCONTINUED标志位可以关注恢复执行的事件，如果有子进程收到SIGCONT信号而恢复执行，waitpid就可以返回。

但是上述两个事件和子进程的终止事件是并列的关系，waitpid成功返回的时候，可能是等到了子进程的终止事件，也可能是等到了暂停或恢复执行的事件。这需要通过status的值来区分。

那么，现在应该分析status的值了。

4.7.4 等待子进程之等待状态值

无论是wait（）函数还是waitpid（）函数，都有一个status变量。这个变量是一个int型指针。可以传递NULL，表示不关心子进程的状态信息。如果不为空，则根据填充的status值，可以获取到子进程的很多信息，如图4-12所示。

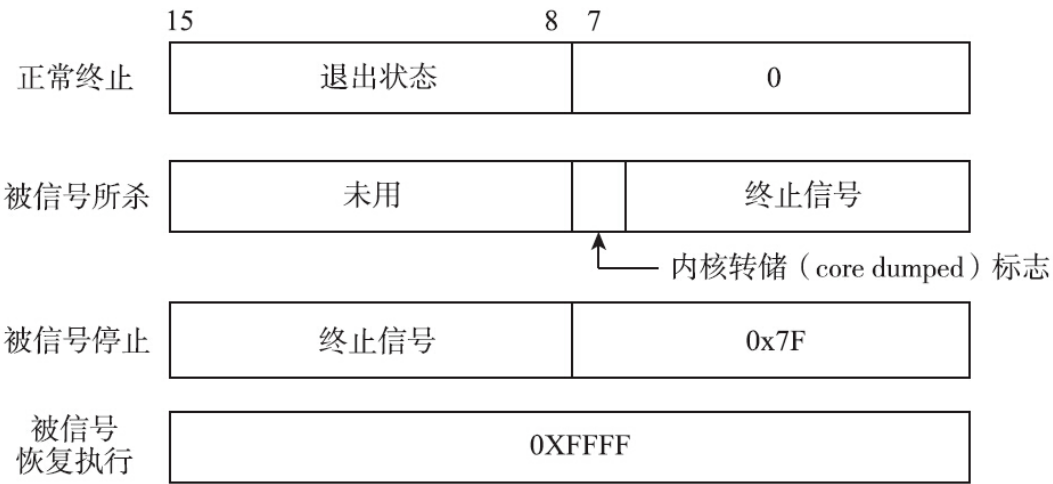


图4-12 wait返回的子进程的状态信息

根据图4-12可知，直接根据status值可以获得进程的退出方式，但是为了保证可移植性，不应该直接解析status值来获取退出状态。因此系统提供了相应的宏（macro），用来解析返回值。下面分别介绍各种情况。

1.进程是正常退出的

有两个宏与正常退出相关，见表4-4。

表4-4 与进程正常退出相关的宏

宏	说 明
WIFEXITED(status)	如果子进程正常退出，则返回 true，否则返回 false
WEXITSTATUS(status)	如果子进程正常退出，则本宏用来获取进程的退出状态

所谓截取退出状态8~15位的值，也就是exit_group系统调用用户传入的int型的值。当然只有最低的8位：

```
#define __WEXITSTATUS(status) (((status) & 0xff00) >> 8)
```

2.进程收到信号，导致退出

有三个宏与这种情况相关，见表4-5。

表4-5 与进程收到信号导致退出相关的宏

宏	说 明
WIFSIGNALED(status)	如果进程是被信号杀死的，则返回 true，否则返回 false
WTREMSIG(status)	如果进程是被信号杀死的，则返回杀死进程的信号的值得
WCOREDUMP(status)	如果子进程产生了 core dump，则返回 true。否则返回 false

3.进程收到信号，被停止

有两个宏与这种情况相关，见表4-6。

表4-6 与进程收到信号被停止相关的宏

宏	说 明
WIFSTOPPED(status)	如果子进程因收到相关信号，暂停执行，处于停止状态，则返回 true，否则返回 fasle
WSTOPSIG(status)	如果子进程处于停止状态，这个宏返回导致子进程停止的信号的值

之所以需要WSTOPSIG宏来返回导致子进程停止的信号值，是因为不只一个信号可以导致子进程停止：SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU，都可以使进程停止。

4.子进程恢复执行

有一个宏与这种情况相关，见表4-7。

表4-7 与子进程恢复执行相关的宏

宏	说 明
WIFCONTINUED(status)	如果由于 SIGCONT 信号的递送，子进程恢复执行，则返回 true，否则返回 false

为何没有返回使子进程恢复的信号值的宏？原因是只有SIGCONT信号能够使子进程从停止状态中恢复过来。如果子进程恢复执行，只可能是收到了SIGCONT信号，所以不需要宏来取信号的值。

下面给出了判断子进程终止的示例代码。等待子进程暂停或恢复执行的情况，可以根据下面的示例代码自行实现。

```
void print_wait_exit(int status)
{
    printf("status = %d\n",status);
    if (WIFEXITED(status))
    {
        printf("normal termination,exit status = %d\n",WEXITSTATUS(status));
    }
    else if (WIFSIGNALED(status))
    {
        printf("abnormal termination,signal number =%d%s\n",WTERMSIG(status),
#ifdef WCOREDUMP
        WCOREDUMP(status)?"core file generated" : "");
#else
        "");
#endif
    }
}
```

尽管waitpid函数对wait函数做了很多的扩展，但waitpid函数还是存在不足之处：

waitpid固然通过WUNTRACE和WCONTINUED标志位，增加了对子进程停止事件和子进程恢复执行事件的支持，但是这种支持并不完美，这两种事件都和子进程的终止事件混在一起了。

wait和waitpid函数都会调用wait4系统调用，无论用户传递的参数为何，总会添上WEXITED事件，如下所示：

```
wo.wo_flags = options | WEXITED;
```

如果用户不关心子进程的终止事件，只关心子进程的停止事件，能否使用waitpid（）明确做到？答案是不行。当waitpid返回时，可能是因为子进程终止，也可能是因为子进程停止。这是waitpid和wait的致命缺陷。

为了解决这个缺陷，wait家族的最重要成员，waitid（）函数就要闪亮登场了。

4.7.5 等待子进程之waitid（）

前面提到过，waitpid函数是wait函数的超集，wait函数能干的事情，waitpid函数都能做到。但是waitpid函数的控制还是不太精确，无论用户是否关心相关子进程的终止事件，终止事件都可能会返回给用户。因此Linux提供了waitid系统调用。glibc封装了waitid系统调用从而实现了waitid函数。尽管目前普遍使用的是wait和waitpid两个函数，但是waitid函数的设计显然更加合理。

waitid函数的接口定义如下：

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

该函数的第一个入参idtype和第二个入参id用于选择用户关心的子进程。

·idtype==P_PID：精确打击，等待进程ID等于id的进程。

·idtype==P_PGID：在所有子进程中等待进程组ID等于id的进程。

·idtype==P_ALL：等待任意子进程，第二个参数id被忽略。

waitid函数的改进在于第四个参数options。options参数是下面标志位的按位或：

·WEXITED：等待子进程的终止事件。

·WSTOPPED：等待被信号暂停的子进程事件。

·WCONTINUED：等待先前被暂停，但是被SIGCONT信号恢复执行的子进程。

这三个标志位互相独立，因此能解决waitpid的致命缺陷，两个函数的标志位关系如表4-8所示。

表4-8 waitpid函数和waitid函数的标志位关系

waitpid 的标志位	等价的 waitid 的标志位
WUNTRACED	WEXITED WSTOPPED
WCONTINUED	WEXITED WCONTINUED

waitid函数还支持其他的标志位。

WNOHANG：这个标志位是老相识了，语义与waitpid一致，与id匹配的子进程若并无状态信息需要返回，则不阻塞，立刻返回，返回值是0。如果调用进程并无子进程与id匹配，则返回-1，并且设置errno为ECHILD。

WNOWAIT：这个标志位是waitid的独门绝技，waitpid和wait函数都不支持。通过前面的讨论可以知道wait并不仅仅是获取子进程的状态信息，它还会改变子进程的状态。最典型的是子进程的退出。wait函数返回之前，子进程处于僵尸状态，取走信息之后，内核负责调用release_task函数来将僵尸子进程的最后残存资源释放掉，子进程彻底消失。WNOWAIT标志位指示内核，只负责获取信息，不要改变子进程的状态。带有WNOWAIT标志位调用waitid函数，稍后还可以调用wait或waitpid或waitid再次获得同样的信息。

第三个参数infop本质是个返回值，系统调用负责将子进程的相关信息填充到infop指向的结构体中。如果成功获取到信息，下面的字段将会被填充：

·si_pid：子进程的进程ID，相当于wait和waitpid成功时的返回值。

·si_uid：子进程真正的用户ID。

·si_signo：该字段总被填成SIGCHLD。

·si_code：指示子进程发生的事件，该字段可能的取值是：

- CLD_EXIT（子进程正常退出）
- CLD_KILLED（子进程被信号杀死）
- CLD_DUMPED（子进程被信号杀死，并且产生了core dump）
- CLD_STOPPED（子进程被信号暂停）
- CLD_CONTINUED（子进程被SIGCONT信号恢复执行）
- CLD_TRAPPED（子进程被跟踪）

·si_status：status值的语义与wait函数及waitpid函数一致。

对于返回值，在两种情况下会返回0：

- 成功等到子进程的变化，并取回相应的信息。
- 设置了WNOHANG标志位，并且子进程状态无变化。

如何区分这两种情况呢？

解决的方法就是判断返回的siginfo_t结构体中的si_pid，如果是由于子进程的状态变化而导致的返回，则si_pid必不等于0，而是等于子进程的进程ID；若子进程状态没有变化，则si_pid等于0。但是标准并没有规定，waitid函数负责将siginfo_t结构体的内容清零，所以为了正确区分这两种情况，唯一安全的做法就是首先将siginfo_t结构体清零，返回后，通过判断si_pid是否为0来分辨这两种情况。示例代码如下：

```
siginfo_t info ;
memset (&info,0,sizeof(siginfo_t));
if (waitid(idtype,id,&info,options | WNOHANG) == -1)
{
    /*发生错误

*/
}
else if(info.si_pid == 0)
{
    /*子进程没有发生变化

*/
}
else
{
    /*若有子进程状态发生变化，则进一步处理之

*/
}
```

4.7.6 进程退出和等待的内核实现

Linux引入多线程之后，为了支持进程的所有线程能够整体退出，内核引入了exit_group系统调用。对于进程而言，无论是调用exit（）函数、_exit（）函数还是在main函数中return，最终都会调用exit_group系统调用。

对于单线程的进程，从do_exit_group直接调用do_exit就退出了。但是对于多线程的进程，如果某一个线程调用了exit_group系统调用，那么该线程在调用do_exit之前，会通过zap_other_threads函数，给每一个兄弟线程挂上一个SIGKILL信号。内核在尝试递送信号给兄弟进程时（通过get_signal_to_deliver函数），会在挂起信号中发现SIGKILL信号。内核会直接调用do_group_exit函数让该线程也退出（如图4-13所示）。这个过程在第3章中已经详细分析过了。

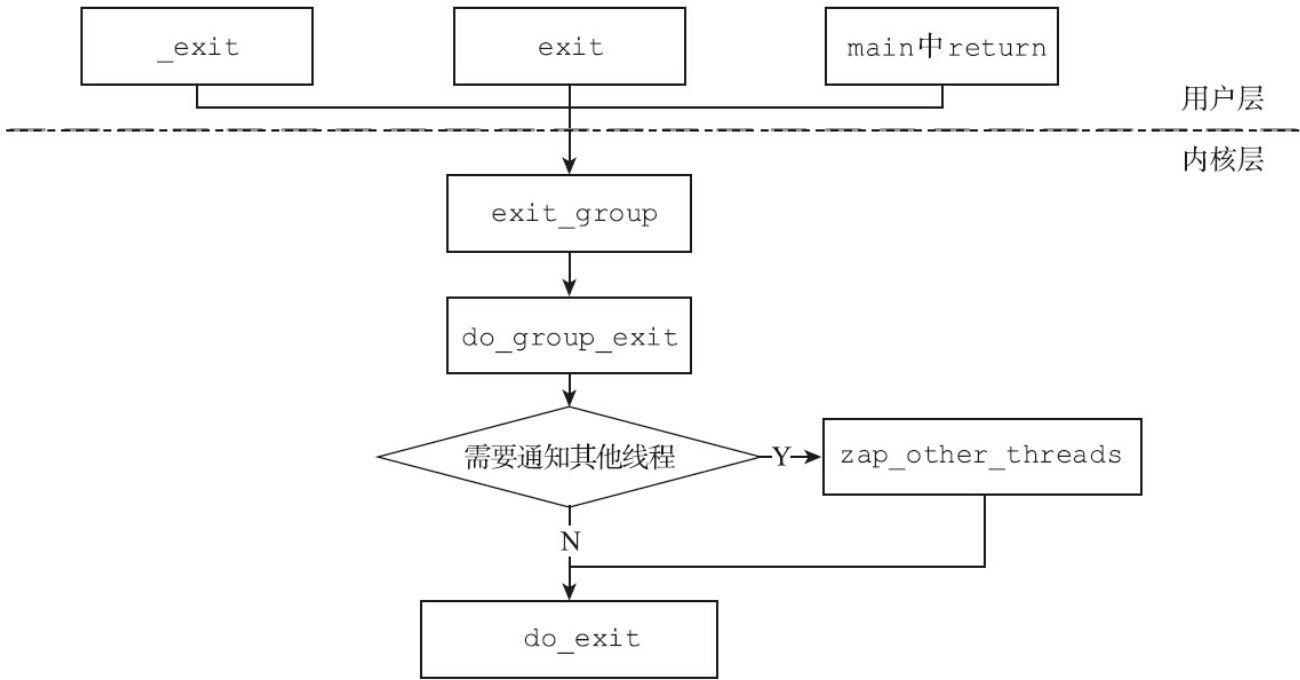


图4-13 进程退出流程图

在do_exit函数中，进程会释放几乎所有的资源（文件、共享内存、信号量等）。该进程并不甘心，因为它还有两桩心愿未了：

- 作为父进程，它可能还有子进程，进程退出以后，将来谁为它的子进程“收尸”。
- 作为子进程，它需要通知它的父进程来为自己“收尸”。

这两件事情是由exit_notify来负责完成的，具体来说forget_original_parent函数和do_notify_parent函数各自负责一件事，如表4-9所示。

表4-9 exit_notify中两个函数及其负责的事务

函 数 名	说 明
forget_original_parent	负责给退出进程的子进程寻找新的父进程。
do_notify_parent	负责通知退出进程的父进程

forget_original_parent（），多么“悲伤”的函数名。顾名思义，该函数用来给自己的子进程安排新的父进程。

给自己的子进程安排新的父进程，细分下来，是两件事情：

- 1) 为子进程寻找新的父进程。
- 2) 将子进程的父进程设置为第1)步中找到的新的父亲。

为子进程寻找父进程，是由find_new_reaper（）函数完成的。如果退出的进程是多线程进程，则可以将子进程托付给自己的兄弟线程。如果

没有这样的线程，就“托孤”给init进程。

为自己的子进程找到新的父亲之后，内核会遍历退出进程的所有子进程，将新的父亲设置为子进程的父亲。

```
static void forget_original_parent(struct task_struct *father)
{
    struct task_struct *p, *n, *reaper;
    LIST_HEAD(dead_children);
    write_lock_irq(&tasklist_lock);
    /*
     * Note that exit_ptrace() and find_new_reaper() might
     * drop tasklist_lock and reacquire it.
     */
    exit_ptrace(father);
    reaper = find_new_reaper(father);
    list_for_each_entry_safe(p, n, &father->children, sibling) {
        struct task_struct *t = p;
        do {
            t->real_parent = reaper;
            if (t->parent == father) {
                BUG_ON(t->ptrace);
                t->parent = t->real_parent;
            }
        } while (t = t->sibling);
        /*内核提供了机制，允许父进程退出时向子进程发送信号*/

        if (t->pdeath_signal)
            group_send_sig_info(t->pdeath_signal,
                                SEND_SIG_NOINFO, t);
        while_each_thread(p, t)
            reparent_leader(father, p, &dead_children);
    }
    write_unlock_irq(&tasklist_lock);
    BUG_ON(!list_empty(&father->children));
    list_for_each_entry_safe(p, n, &dead_children, sibling) {
        list_del_init(&p->sibling);
        release_task(p);
    }
}
```

这部分代码比较容易引起困扰的是下面这行，我们都知道，子进程“死”的时候，会向父进程发送信号SIGCHLD，Linux也提供了一种机制，允许父进程“死”的时候向子进程发送信号。

```
if (t->pdeath_signal)
    group_send_sig_info(t->pdeath_signal,
                        SEND_SIG_NOINFO, t);
```

读者可以通过man prctl，查看PR_SET_PDEATHSIG标志位部分。如果应用程序通过prctl函数设置了父进程“死”时要向子进程发送信号，就会执行到这部分内核代码，以通知其子进程。

接下来是第二桩未了的心愿：想办法通知父进程为自己“收尸”。

对于单线程的程序来说完成这桩心愿比较简单，但是多线程的情况就复杂些。只有线程组的主线程才有资格通知父进程，线程组的其他线程终止的时候，不需要通知父进程，也没必要保留最后的资源并陷入僵尸态，直接调用release_task函数释放所有资源就好。

为什么要这样设计？细细想来，这么做是合理的。父进程创建子进程时，只有子进程的主线程是父进程亲自创建出来的，是父进程的亲生儿子，父进程也只关心它，至于子进程调用pthread_create产生的其他线程，父进程压根就不关心。

由于父进程只认子进程的主线程，所以在线程组中，主线程一定要挺住。在用户层面，可以调用pthread_exit让主线程先“死”，但是在内核态中，主线程的task_struct一定要挺住，哪怕变成僵尸，也不能释放资源。

生命在于“折腾”，如果主线程率先退出了，而其他线程还在正常工作，内核又将如何处理？

```
else if (thread_group_leader(tsk)) {
    /*线程组组长只有在全部线程都已退出的情况下，

    *才能调用

do_notify_parent通知父进程

*/
    autoreap = thread_group_empty(tsk) &&
do_notify_parent(tsk, tsk->exit_signal);
} else {
    /*如果是线程组的非组长线程，可以立即调用

release_task.
```

*释放残余的资源，因为通知父进程这件事和它没有关系

```
*/
    autoreap = true;
}
```

上面的代码给出了答案，如果退出的进程是线程组的主线程，但是线程组中还有其他线程尚未终止（thread_group_empty函数返回false），那么autoreaper就等于false，也就不会调用do_notify_parent向父进程发送信号了。

因为子进程的线程组中有其他线程还活着，因此子进程的主线程退出时不能通知父进程，错过了调用do_notify_parent的机会，那么父进程如何才能知晓子进程已经退出了呢？答案会在最后一个线程退出时揭晓。此答案就藏在内核的release_task函数中：

```
leader = p->group_leader;
if (leader != p && thread_group_empty(leader) && leader->exit_state == EXIT_ZOMBIE) {
    zap_leader = do_notify_parent(leader, leader->exit_signal);
    if (zap_leader)
        leader->exit_state = EXIT_DEAD;
}
```

当线程组的最后一个线程退出时，如果发现：

- 该线程不是线程组的主线程。
- 线程组的主线程已经退出，且处于僵尸状态。
- 自己是最后一个线程。

同时满足这三个条件的时候，该子进程就需要冒充线程组的组长，即以子进程的主线程的身份来通知父进程。

上面讨论了一种比较少见又比较折腾的场景，正常的多线程编程应该不会如此安排。对于多线程的进程，一般情况下会等所有其他线程退出后，主线程才退出。这时，主线程会在exit_notify函数中发现自己是组长，线程组里所有成员均已退出，然后它调用do_notify_parent函数来通知父进程。

无论怎样，子进程都走到了do_notify_parent函数这一步。该函数是完成父子进程之间互动的主要函数。

```
bool do_notify_parent(struct task_struct *tsk, int sig)
{
    struct siginfo info;
    unsigned long flags;
    struct sighand_struct *psig;
    bool autoreap = false;
    BUG_ON(sig == -1);
    /* do_notify_parent_cldstop should have been called instead. */
    BUG_ON(task_is_stopped_or_traced(tsk));
    BUG_ON(!tsk->ptrace &&
            (tsk->group_leader != tsk || !thread_group_empty(tsk)));
    if (sig != SIGCHLD) {
        /*
         * This is only possible if parent == real_parent.
         * Check if it has changed security domain.
         */
        if (tsk->parent_exec_id != tsk->parent->self_exec_id)
            sig = SIGCHLD;
    }
    info.si_signo = sig;
    info.si_errno = 0;
    rcu_read_lock();
    info.si_pid = task_pid_nr_ns(tsk, tsk->parent->nsproxy->pid_ns);
    info.si_uid = __task_cred(tsk)->uid;
    rcu_read_unlock();
    info.si_utime = cputime_to_clock_t(cputime_add(tsk->utime,
            tsk->signal->utime));
    info.si_stime = cputime_to_clock_t(cputime_add(tsk->stime,
            tsk->signal->stime));
    info.si_status = tsk->exit_code & 0x7f;
    if (tsk->exit_code & 0x80)
        info.si_code = CLD_DUMPED;
    else if (tsk->exit_code & 0x7f)
        info.si_code = CLD_KILLED;
    else {
        info.si_code = CLD_EXITED;
        info.si_status = tsk->exit_code >> 8;
    }
    psig = tsk->parent->sighand;
    spin_lock_irqsave(&psig->siglock, flags);
    if (!tsk->ptrace && sig == SIGCHLD &&
        (psig->action[SIGCHLD-1].sa.sa_handler == SIG_IGN ||
         (psig->action[SIGCHLD-1].sa.sa_flags & SA_NOCLDWAIT))) {
        autoreap = true;
        if (psig->action[SIGCHLD-1].sa.sa_handler == SIG_IGN)
            sig = 0;
    }
    /*子进程向父进程发送信号
```

```
*/
    if (valid_signal(sig) && sig)
        __group_send_sig_info(sig, &info, tsk->parent);
    /* 子进程尝试唤醒父进程，如果父进程正在等待其终止
```

```
*/_wake_up_parent(tsk, tsk->parent);
spin_unlock_irqrestore(&psig->siglock, flags);
return autoreap;
}
```

父子进程之间的互动有两种方式：

- 子进程向父进程发送信号SIGCHLD。
- 子进程唤醒父进程。

对于这两种方法，我们分别展开讨论。

1.父子进程互动之SIGCHLD信号

父进程可能并不知道子进程是何时退出的，如果调用wait函数等待子进程退出，又会导致父进程陷入阻塞，无法执行其他任务。那有没有一种办法，让子进程退出的时候，异步通知到父进程呢？答案是肯定的。当子进程退出时，会向父进程发送SIGCHLD信号。

父进程收到该信号，默认行为是置之不理。在这种情况下，子进程就会陷入僵尸状态，而这又会浪费系统资源，该状态会维持到父进程退出，子进程被init进程接管，init进程会等待僵尸进程，使僵尸进程释放资源。

如果父进程不太关心子进程的退出事件，听之任之可不是好办法，可以采取以下办法：

- 父进程调用signal函数或sigaction函数，将SIGCHLD信号的处理函数设置为SIG_IGN。
- 父进程调用sigaction函数，设置标志位时置上SA_NOCLDWAIT位（如果不关心子进程的暂停和恢复执行，则置上SA_NOCLDSTOP位）。

从内核代码来看，如果父进程的SIGCHLD的信号处理函数为SIG_IGN或sa_flags中被置上了SA_NOCLDWAIT位，子进程运行到此处时就知道了，父进程并不关心自己的退出信息，do_notify_parent函数就会返回true。在外层的exit_notify函数发现返回值是true，就会调用release_task函数，释放残余的资源，自行了断，子进程也就不会进入僵尸状态了。

如果父进程关心子进程的退出，情况就不同了。父进程除了调用wait函数之外，还有了另外的选择，即注册SIGCHLD信号处理函数，在信号处理函数中处理子进程的退出事件。

为SIGCHLD写信号处理函数并不简单，原因是SIGCHLD是传统的不可靠信号。信号处理函数执行期间，会将引发调用的信号暂时阻塞（除非显式地指定了SA_NODEFER标志位），在这期间收到的SIGCHLD之类的传统信号，都不会排队。因此，如果在处理SIGCHLD信号时，有多个子进程退出，产生了多个SIGCHLD信号，但父进程只能收到一个。如果在信号处理函数中，只调用一次wait或waitpid，则会造成某些僵尸进程成为漏网之鱼。

正确的写法是，信号处理函数内，带着NOHANG标志位循环调用waitpid。如果返回值大于0，则表示不断等待子进程退出，返回0则表示，当前没有僵尸子进程，返回-1则表示出错，最大的可能就是errno等于ECHLD，表示所有子进程都已退出。

```
while(waitpid(-1,&status,WNOHANG) > 0)
{
    /*此处处理返回信息*/

    */
    continue;
}
```

信号处理函数中的waitpid可能会失败，从而改变全局的errno的值，当主程序检查errno时，就有可能发生冲突，所以进入信号处理函数前要现保存errno到本地变量，信号处理函数退出前，再恢复errno。

2.父子进程互动之等待队列

上一种方法可以称之为信号通知。另一种情况是父进程调用wait主动等待。如果父进程调用wait陷入阻塞，那么子进程退出时，又该如何及时唤醒父进程呢？

前面提到了，子进程会调用__wake_up_parent函数，来及时唤醒父进程。事实上，前提条件是父进程确实在等待子进程的退出。如果父进程并没有调用wait系列函数等待子进程的退出，那么，等待队列为空，子进程的__wake_up_parent对父进程并无任何影响。

```
void __wake_up_parent(struct task_struct *p, struct task_struct *parent)
{
    __wake_up_sync_key(&parent->signal->wait_chldexit,
        TASK_INTERRUPTIBLE, 1, p);
}
```

父进程的进程描述符的signal结构体中有wait_chldexit变量，这个变量是等待队列头。父进程调用wait系列函数时，会创建一个wait_opts结构体，并把该结构体挂入等待队列中。

```
static long do_wait(struct wait_opts *wo)
{
    struct task_struct *tsk;
    int retval;
    trace_sched_process_wait(wo->wo_pid);
    /*挂入等待队列

*/
    init_waitqueue_func_entry(&wo->child_wait, child_wait_callback);
    wo->child_wait.private = current;
    add_wait_queue(&current->signal->wait_chldexit, &wo->child_wait);
repeat:
    /**/
    wo->notask_error = -ECHILD;
    if ((wo->wo_type < PIDTYPE_MAX) &&
        (!wo->wo_pid || hlist_empty(&wo->wo_pid->tasks[wo->wo_type])))
        goto notask;
    set_current_state(TASK_INTERRUPTIBLE);
    read_lock(&tasklist_lock);
    tsk = current;
    do {
        retval = do_wait_thread(wo, tsk);
        if (retval)
            goto end;
        retval = ptrace_do_wait(wo, tsk);
        if (retval)
            goto end;
        if (wo->wo_flags & __WNOTHREAD)
            break;
    } while (each_thread(current, tsk));
    read_unlock(&tasklist_lock);
    /*找了一圈，没有找到满足等待条件的的子进程，下一步的行为将取决于

WNOHANG标志位

*如果将

WNOHANG标志位置位，则表示不等了，直接退出。

*如果没有置位，则让出

CPU，醒来后继续再找一圈

*/
notask:
    retval = wo->notask_error;
    if (!retval && !(wo->wo_flags & WNOHANG)) {
        retval = -ERESTARTSYS;
        if (!signal_pending(current)) {
            schedule();
            goto repeat;
        }
    }
end:
    set_current_state(TASK_RUNNING);
    remove_wait_queue(&current->signal->wait_chldexit, &wo->child_wait);
    return retval;
}
```

父进程先把自己设置成TASK_INTERRUPTIBLE状态，然后开始寻找满足等待条件的子进程。如果找到了，则将自己重置成TASK_RUNNING状态，欢乐返回；如果没找到，就要根据WNOHANG标志位来决定等不等待子进程。如果没有WNOHANG标志位，那么，父进程就会让出CPU资源，等待别人将它唤醒。

回到另一头，子进程退出的时候，会调用__wake_up_parent，唤醒父进程，父进程醒来以后，回到repeat，再次扫描。这样做，子进程的退出就能及时通知到父进程，从而使父进程的wait系列函数可以及时返回。

4.8 exec家族

前面讨论了进程的创建和退出，`exec`家族函数在其中犹抱琵琶半遮面，现在是时候让`exec`家族函数登台亮相了。

整个`exec`家族有6个函数，这些函数都是构建在`execve`系统调用之上的。该系统调用的作用是，将新程序加载到进程的地址空间，丢弃旧有的程序，进程的栈、数据段、堆栈等会被新程序替换。

基于`execve`系统调用的6个`exec`函数，接口虽然各异，实现的功能却是相同的，首先我们来讲述与系统调用同名的`execve`函数。

4.8.1 execve函数

execve函数的接口定义如下：

```
#include <unistd.h>
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

其中，参数filename是准备执行的新程序的路径名，可以是绝对路径，也可以是相对于当前工作目录的相对路径。

后面的第二个参数很容易让我们联想到C语言的main（）函数的第二个参数，事实上格式也是一样的：字符串指针组成的数组，以NULL结束。argv[0]一般对应可执行文件的文件名，也就是filename中的basename（路径名最后一个/后面的部分）。当然如果argv[0]不遵循这个约定也无妨，因为execve可以从第一个参数获取到要执行文件的路径，只要不是NULL即可。

第三个参数与C语言的main函数中的第三个参数envp一样，也是字符串指针数组，以NULL结束，指针指向的字符串的格式为name=value。

一般来说，execve（）函数总是紧随fork函数之后。父进程调用fork之后，子进程执行execve函数，抛弃父进程的程序段，和父进程分道扬镳，从此天各一方，各走各路。但是也可以不执行fork，单独调用execve函数：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    char *args[] = {"/bin/ls", "-l", NULL};
    if(execve("/bin/ls", args, NULL) == -1) {
        perror("execve");
        exit(EXIT_FAILURE);
    }
    puts("Never get here");
    exit(EXIT_SUCCESS);
}
```

本着“贵在折腾”的原则，上面写了一个不fork直接调用execve的程序。调用execve后，程序就变成了/bin/sh-l。这个程序的输出如下：

```
total 16
-rwxr-xr-x 1 root root 8672 Dec 27 20:40 exec_no_fork
-rw-r--r-- 1 root root 288 Dec 27 20:40 exec_no_fork.c
```

我们可以看到，代码段最后的Never get here没有被打印出来，这是因为execve函数的返回是特殊的。如果失败，则会返回-1，但是如果成功，则永不返回，这是可以理解的。execve做的就是斩断过去，奔向新生活的事情，如果成功，自然不可能再返回来，再次执行老程序的代码。

所以无须检查execve的返回值，只要返回，就必然是-1。可以从errno判断出出错的原因。出错的可能性非常多，手册提供了19种不同的errno，罗列了22种失败的情景。很难记住，好在大部分都不常见，常见的情况有以下几种：

- EACCESS：这个是我们最容易想到的，就是第一个参数filename，不是个普通文件，或者该文件没有赋予可执行的权限，或者目录结构中某一级目录不可搜索，或者文件所在的文件系统是以MS_NOEXEC标志挂载的。

- ENOENT：文件不存在。

- ETXTBSY：存在其他进程尝试修改filename所指代的文件。

- ENOEXEC：这个错误其实是比较高端的一种错误了，文件存在，也可以执行，但是无法执行，比如说，Windows下的可执行程序，拿到Linux下，调用execve来执行，文件的格式不对，就会返回这种错误。

上面提到的ENOEXEC错误码，其实已经触及了execve函数的核心，即哪些文件是可以执行的，execve系统调用又是如何执行的呢？这些会在execve系统调用的内核系统调用中详细介绍。

4.8.2 exec家族

- 从内核的角度来说，提供execve系统调用就足够了，但是从应用层编程的角度来讲，execve函数就并不那么好使了：
- 第一个参数必须是绝对路径或是相对于当前工作目录的相对路径。习惯在shell下工作的用户会觉得不太方便，因为日常工作都是写ls和mkdir之类命令的，没有人会写/bin/ls或/bin/mkdir。shell提供了环境变量PATH，即可执行程序的查找路径，对于位于查找路径里的可执行程序，我们不必写出完整的路径，很方便，而execve函数享受不到这个福利，因此使用不便。
- execve函数的第三个参数是环境变量指针数组，用户使用execve编程时不得不自己负责环境变量，书写大量的“key=value”，但大部分情况下并不需要定制环境变量，只需要使用当前的环境变量即可。

正是为了提供相应的便利，所以用户层提供了6个函数，当然，这些函数本质上都是调用execve系统调用，只是使用的方法略有不同，代码如下：

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg,
            ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[],
           char *const envp[]);
```

上述6个函数分成上下两个半区。分类的依据是参数采用列表（l，表示list）还是数组（v，表示vector）。上半区采用列表，它们会罗列所有的参数，下半区采用数组。

在每个半区之中，带p的表示可以使用环境变量PATH，带e的表示必须要自己维护环境变量，而不使用当前环境变量，具体见表4-10。

表4-10 exec家族函数

函 数 名	参 数 格 式	是否自动搜索 PATH	是否使用当前环境变量
execl	列表	不是	是
execlp	列表	是	是
execl_e	列表	不是	不是，须自己组装环境变量
execv	数组	不是	是
execvp	数组	是	是
execve	数组	不是	不是，须自己组装环境变量

举个例子来加深记忆：

```
#include <unistd.h>
char *const ps_argv[] = {"ps", "-ax", NULL};
char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console", NULL};
execl("/bin/ps", "ps", "-ax", NULL);
/*带

p的，可以使用环境变量

PATH，无须写全路径

*/
execlp("ps", "ps", "-ax", NULL);
/*带

e的需要自己组装环境变量

*/
execl_e("/bin/ps", "ps", "-ax", NULL, ps_envp);
execv("/bin/ps", ps_argv);
```

```
/*带
```

p的, 可以使用环境变量

PATH, 无须写全路径

```
*/  
execvp("ps",ps_argv);  
/*带
```

e的需要自己组拼环境变量

```
*/  
execve("/bin/ps",ps_argv,ps_envp);
```

4.8.3 execve系统调用的内核实现

前面提到的ENOEXEC错误表示内核不知道如何执行对应的可执行文件。Linux支持很多种可执行文件的格式，有渐渐退出历史舞台的a.out格式，有比较通用的ELF格式的文件，还有shell脚本文件、python脚本、java文件、php文件等。对于这些形形色色的可执行文件，内核该如何正确地执行呢？直接将Windows平台上的可执行文件拷贝到Linux下，Linux为什么不能执行（假设没有wine这个执行Windows程序的工具）？这是本节需要解决的问题。要解决上述问题，首先还是需要深入内核。

execve是平台相关的系统调用，刨去我们不太关心的平台差异，内核都会走到do_execve_common函数这一步。

```
static int do_execve_common(const char *filename,
                           struct user_arg_ptr argv,
                           struct user_arg_ptr envp,
                           struct pt_regs *regs)
{
    struct linux_binprm *bprm;
    struct file *file;
    struct files_struct *displaced;
    bool clear_in_exec;
    int retval;
    const struct cred *cred = current_cred();
    if ((current->flags & PF_NPROC_EXCEEDED) &&
        atomic_read(&cred->user->processes) > rlimit(RLIMIT_NPROC)) {
        retval = -EAGAIN;
        goto out_ret;
    }
    /* We're below the limit (still or again), so we don't want to make
     * further execve() calls fail. */
    current->flags &= ~PF_NPROC_EXCEEDED;
    retval = unshare_files(&displaced);
    if (retval)
        goto out_ret;
    retval = -ENOMEM;
    bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
    if (!bprm)
        goto out_files;
    retval = prepare_bprm_creds(bprm);
    if (retval)
        goto out_free;
    retval = check_unsafe_exec(bprm);
    if (retval < 0)
        goto out_free;
    clear_in_exec = retval;
    current->in_execve = 1;
    /*读取可执行文件

*/
    file = open_exec(filename);
    retval = PTR_ERR(file);
    if (IS_ERR(file))
        goto out_unmark;
    /*选择负载最小的

CPU来执行新程序

*/
    sched_exec();
    bprm->file = file;
    bprm->filename = filename;
    bprm->interp = filename;
    retval = bprm_mm_init(bprm);
    if (retval)
        goto out_file;
    bprm->argc = count(argv, MAX_ARG_STRINGS);
    if ((retval = bprm->argc) < 0)
        goto out;
    bprm->envc = count(envp, MAX_ARG_STRINGS);
    if ((retval = bprm->envc) < 0)
        goto out; /*填充

linux_binprm数据结构

*/
    retval = prepare_binprm(bprm);
    if (retval < 0)
        goto out;
    /*接下来的
```

3个

copy用来拷贝文件名、命令行参数和环境变量

```
*/
    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0)
        goto out;
    bprm->exec = bprm->p;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0)
        goto out;
    retval = copy_strings(bprm->argc, argv, bprm);
    if (retval < 0)
        goto out;
    /*核心部分，遍历
```

formats链表，尝试每个

load_binary函数

```
*/retval = search_binary_handler(bprm, regs);
    if (retval < 0)
        goto out;
    /* execve succeeded */
    current->fs->in_exec = 0;
    current->in_execve = 0;
    acct_update_integrals(current);
    free_bprm(bprm);
    if (displaced)
        put_files_struct(displaced);
    return retval;
out:
    if (bprm->mm) {
        acct_arg_size(bprm, 0);
        mmput(bprm->mm);
    }
out_file:
    if (bprm->file) {
        allow_write_access(bprm->file);
        fput(bprm->file);
    }
out_unmark:
    if (clear_in_exec)
        current->fs->in_exec = 0;
    current->in_execve = 0;
out_free:
    free_bprm(bprm);
out_files:
    if (displaced)
        reset_files_struct(displaced);
out_ret:
    return retval;
}
```

其中，linux_binprm是重要的结构体，它与稍后提到的linux_binfmt联手，支持了Linux下多种可执行文件的格式。首先，内核会将程序运行需要的参数argv和环境变量搜集到linux_binprm结构体中，比较关键的一步是：

```
retval = prepare_binprm(bprm);
```

在prepare_binprm函数中读取可执行文件的头128个字节，存放在linux_binprm结构体的buf[BINPRM_BUF_SIZE]中。我们知道日常写shell脚本、python脚本的时候，总是会在第一行写下如下语句：

```
#!
```

```
/bin/bash
#! /usr/bin/python
#!
```

```
/usr/bin/env python
```

开头的#! 被称为shebang，又被称为sha-bang、hashbang等，指的就是脚本中开始的字符。在类Unix操作系统中，运行这种程序，需要相应的解释器。使用哪种解释器，取决于shebang后面的路径。#! 后面跟随的一般是解释器的绝对路径，或者是相对于当前工作目录的相对路径。格式如下所示：

```
#! interpreter [optional-arg]
```

解释器是绝对路径或是相对于当前工作目录的相对路径，这就给脚本的可移植性带来了挑战。以python的解释器为例，python可能位于/usr/bin/python，也可能位于/usr/local/bin/python，甚至有的还位于/home/username/bin/python。这样编写的脚本在新的环境里面运行时，用户就不得不修改脚本了，当大量的脚本移植到新环境中运行时，修改量是巨大的。为了解决这个问题，系统又引入了如下格式：

```
#!/usr/bin/env python
```

在执行时，这种格式会从环境变量\$PATH中查找python解释器。如果存在多个版本的解释器，则会按照\$PATH中查找路径的顺序来查找。

```
manu@manu-hacks:~$ echo $PATH
/home/manu/bin:/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

如果执行方式是./python_script的方式，就会优先查找/home/manu/bin/python，/usr/local/bin/python次之.....如下所示：

```
execve("/home/manu/bin/python", ["python", "./hello.py"], [/ 25 vars *]) = -1 ENOENT (No such file or directory)
execve("/usr/local/bin/python", ["python", "./hello.py"], [/ 25 vars *]) = -1 ENOENT (No such file or directory)
execve("/usr/local/sbin/python", ["python", "./hello.py"], [/ 25 vars *]) = -1 ENOENT (No such file or directory)
execve("/usr/local/bin/python", ["python", "./hello.py"], [/ 25 vars *]) = -1 ENOENT (No such file or directory)
execve("/usr/sbin/python", ["python", "./hello.py"], [/ 25 vars *]) = -1 ENOENT (No such file or directory)
execve("/usr/bin/python", ["python", "./hello.py"], [/ 25 vars *]) = 0
```

上面提到的是脚本文件，除此以外，还有其他格式的文件。Linux平台上最主要的可执行文件格式是ELF格式，当然还有出现较早，逐渐退出历史舞台的a.out格式，这些文件的特点是最初的128字节中都包含了可执行文件的属性的重要信息。比如图4-14中ELF格式的可执行文件，开头4字节为7F 45（E）4C（L）46（F）。

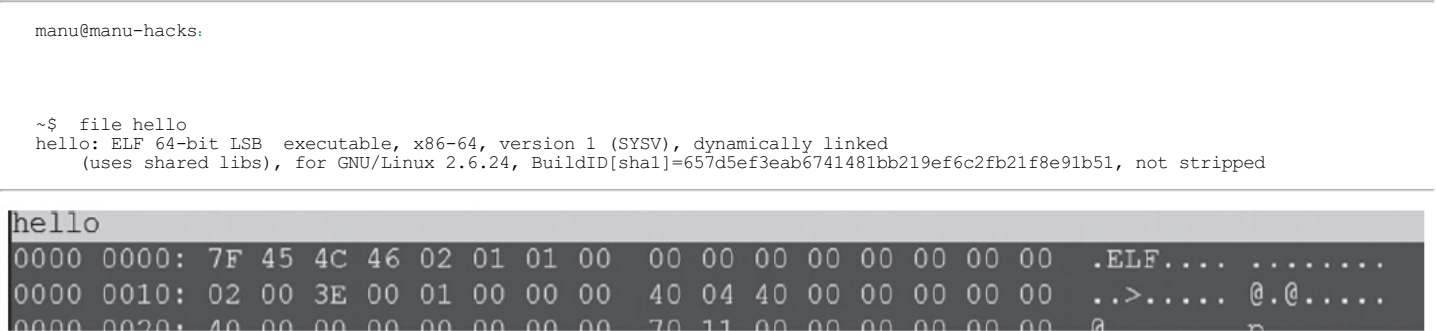


图4-14 ELF文件的头部信息

prepare_binprm函数将文件开始的128字节存入linux_binprm，是为了让后面的程序根据文件开头的magic number选择正确的处理方式。

做完准备工作后，开始执行，核心代码位于search_binary_handler（）函数中。内核之中存在一个全局链表，名叫formats，挂到此链表的数据结构为struct linux_binfmt:

```
struct linux_binfmt {
    struct list_head lh;
    struct module *module;
    int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
    int (*load_shlib)(struct file *);
    int (*core_dump)(struct coredump_params *cprm);
    unsigned long min_coredump; /* minimal dump size */
};
```

操作系统启动的时候，每个编译进内核的可执行文件的“代理人”都会调用register_binfmt函数来注册，把自己挂到formats链表中。每个成员代表一种可执行文件的代理人，前面提到过，会将可执行文件的头128字节存放到linux_binprm的buf中，同时会将运行时的参数和环境变量也存放到linux_binprm的相关结构中。formats链表中的成员依次前来认领，如果是自己代表的可执行文件的格式，后面执行的事情，就委托给了该“代理人”。

如果遍历了链表，所有的linux_binfmt都表示不认识该可执行文件，那又当如何呢？这种情况要根据头部的信息，查看是否有为该格式设计的，作为可动态安装的模块实现的“代理人”存在。如果有的话，就把该模块安装进来，挂入全局的formats链表之中，然后让formats链表中的所有成员再试一次。

上述逻辑位于search_binary_handler函数之中：

```
int search_binary_handler(struct linux_binprm *bprm, struct pt_regs *regs)
{
    unsigned int depth = bprm->recursion_depth;
    int try, retval;
    struct linux_binfmt *fmt;
    pid_t old_pid;
    /* This allows 4 levels of binfmt rewrites before failing hard. */
    if (depth > 5)
        return -ELOOP;
    retval = security_bprm_check(bprm);
    if (retval)
        return retval;
    retval = audit_bprm(bprm);
    if (retval)
        return retval;
    /* Need to fetch pid before load_binary changes it */
    rcu_read_lock();
    old_pid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
    rcu_read_unlock();
    retval = -ENOENT;
    /*最多尝试两次，第一次遍历
```

formats链表中的所有成员，

*若没找到，则尝试加载动态模块，再次遍历

```
*/
for (try=0; try<2; try++) {
    read_lock(&binfmt_lock);
    list_for_each_entry(fmt, &formats, lh) {
        int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
        if (!fn)
            continue;
        if (!try_module_get(fmt->module))
            continue;
        read_unlock(&binfmt_lock);
        bprm->recursion_depth = depth + 1;
        retval = fn(bprm, regs);
        bprm->recursion_depth = depth;
        if (retval >= 0) {
            if (depth == 0)
                ptrace_event(PTRACE_EVENT_EXEC,
                             _old_pid);
            put_binfmt(fmt);
            allow_write_access(bprm->file);
            if (bprm->file)
                fput(bprm->file);
            bprm->file = NULL;
            current->did_exec = 1;
            proc_exec_connector(current);
            return retval;
        }
    }
    read_lock(&binfmt_lock);
```

```
        put_binfmt(fmt);
        if (retval != -ENOEXEC || bprm->mm == NULL)
            break;
        if (!bprm->file) {
            read_unlock(&binfmt_lock);
            return retval;
        }
        read_unlock(&binfmt_lock);
#ifdef CONFIG_MODULES
        if (retval != -ENOEXEC || bprm->mm == NULL) {
            break;
        } else {
#define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))
            if (printable(bprm->buf[0]) &&
                printable(bprm->buf[1]) &&
                printable(bprm->buf[2]) &&
                printable(bprm->buf[3]))
                break; /* -ENOEXEC */
            if (try)
                break; /* -ENOEXEC */
            request_module("binfmt-%04x", *(unsigned short *)(&bprm->buf[2]));
        }
#else
        break;
#endif
    }
    return retval;
}
```

我们可以通过下面的方式来查看自己机器的编译选项，从而得知支持的可执行文件的类型：

```
grep BINFMT /boot/config-3.13.0-43-generic
CONFIG_BINFMT_ELF=y
CONFIG_COMPAT_BINFMT_ELF=y
CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE=y
CONFIG_BINFMT_SCRIPT=y
CONFIG_BINFMT_MISC=m
```

在内核代码树中fs目录下，Makefile记录了支持的格式，在fs目录下，每一种支持的格式xx都有一个binfmt_xx.c文件。

binfmt_aout.c是对应a.out类型的可执行文件，这种文件格式是早期Unix系统使用的可执行文件的格式，由AT&T设计，今天已经退出了历史舞台。

binfmt_elf.c对应的是ELF格式的可执行文件。ELF最早由Unix系统实验室（Unix SYSTEM Laboratories USL）开发，目的是取代传统的a.out格式。1994年6月ELF格式出现在Linux系统上，目前，ELF格式已经成为Linux下最主要的可执行文件格式。

binfmt_script对应的是script格式的可执行文件，这种格式的可执行文件一般以“#!”开头，查找相应的解释器来执行脚本。比如python脚本、shell脚本和perl脚本等。

早期的内核之中，曾经为Java格式提供了专门的binfmt结构，后来取消了，原因是Java并不特殊，不值得为其提供专门的binfmt结构。如果专门为Java提供了，其他语言就会有意见了，没有做到一视同仁。但是需要支持的可执行文件的格式越来越多，大家都可能有自己的解释器，内核支持也不可能无限地增加binfmt结构，这时候，binfmt_misc就出现了。binfmt把这个功能开放给了用户层，用户可以引入自己的可执行文件格式，只要你能定义好magic number，识别出文件是不是自己的这种格式，另外自己定义好解释器就可以了。

binfmt_misc这个机制非常好，提供了支持额外可执行格式的可扩展方法。举例来讲，如果想在Linux下执行Windows的.exe文件，Wine软件可以在Linux下执行Windows的.exe文件。

```
wine application.exe
```

我们可以将Windows exe文件注册到binfmt_misc，直接使用如下方法即可执行exe文件：

```
./application.exe
```

方法就是：

```
echo ':Wine:M::MZ::usr/bin/wine:' > /proc/sys/fs/binfmt_misc/register
```

如果`/proc/sys/fs/binfmt_misc`目录并不存在，则表明`binfmt_misc`并没挂载，那就需要：

```
mount -t binfmt_misc binfmt_misc /proc/sys/fs/binfmt_misc
```

或者在`/etc/fstab`中添加如下行：

```
binfmt_misc /proc/sys/fs/binfmt_misc binfmt_misc defaults 0 0
```

注册某种可执行文件到`binfmt_misc`的格式时，`echo`的内容如下所示：

```
:Name:Type:Offset:String:Mask:Interpreter:Flags
```

其中各个字段的含义是：

- Name:** 产生在`/proc/sys/fs/binfmt_misc`目录下的文件名，代表一种可执行文件。
- Type:** 表示识别类型，M表示用magic number来识别，E表示扩展。
- Offset:** magic number数在文件中的起始偏移量。
- String:** 以magic number或以扩展名匹配的字符串。
- Mask:** 用来屏蔽String中的一些位的字符串。
- Interpret:** 解释程序的完整路径名。
- Flags:** 可选标志，控制必须怎样调用解释程序。

根据这个解释，我们`echo`语句的含义是：Windows可执行文件的前两个字节是magic number，值为MZ，由解释程序`/usr/bin/wine`执行这个可执行文件。

从表面来看，有很多种类型的文件，但是最终都会归结到ELF格式，这是因为那些脚本的解释器是ELF格式。限于篇幅，这里就不介绍内核如何加载执行ELF格式的可执行程序了。毛德操前辈的《Linux内核情景分析》一书中详细分析了a.out类型的可执行文件的加载执行；王柏生前辈的《深入探索Linux操作系统》一书中详细介绍了ELF类型的可执行文件的加载执行，感兴趣的朋友可以参看其中的内容。

4.8.4 exec与信号

exec系列函数，会将现有进程的所有文本段抛弃，直接奔向新生活。调用exec之前，进程可能执行过signal或sigaction，为某些信号注册了新的信号处理函数。一旦决裂，这些新的信号处理函数就无处可寻了。所以内核会为那些曾经改变信号处理函数的信号负责，将它们的处理函数重新设置为SIG_DFL。

这里有一个特例，就是将处理函数设置为忽略（SIG_IGN）的SIGCHLD信号。调用exec之后，SIGCHLD的信号处理函数是保持为SIG_IGN还是重置成SIG_DFL，SUSv3语焉不详，这点要取决于操作系统。对于Linux系统而言，采用的是前者：保持为SIG_IGN。

4.8.5 执行exec之后进程继承的属性

执行exec的进程，其个性虽然叛逆，与过去做了决裂，但是也继承了过去的一些属性。exec运行之后，与进程相关的ID都保持不变。如果进程在执行exec之前，设置了告警（如调用了alarm函数），那么在告警时间到时，它仍然会产生一个信号。在执行exec后，挂起信号依然保留。创建文件时，掩码umask和执行exec之前一样。表4-11给出了执行exec之后进程继承的属性。

表4-11 调用exec之后进程保持的属性

属 性	相关的函数	属 性	相关的函数
进程 ID	getpid	根目录	
父进程 ID	getppid	文件模式创建掩码	umask
进程组 ID	getpgid	文件锁和记录锁	flock 和 fcntl
会话 ID	getsid	进程信号屏蔽	sigprocmask
控制终端	tcgetpgrp	进程挂起的信号	sigpending
真实用户 ID	getsid	已用的时间	times
真实组 ID	getgid	资源限制	getrlimit、setrlimit
附加组 ID	getgroups	nice 值	nice
告警剩余时间	alarm	semadj 值	semop
当前工作目录	getcwd		

通过fork创建的子进程继承的属性和执行exec之后进程保持的属性，两相比较，差异不小。对于fork而言：

- 告警剩余时间：不仅仅是告警剩余时间，还有其他定时器（setitimer、timer_create等），fork创建的子进程都不继承。
- 进程挂起信号：子进程会将挂起信号初始化为空。
- 信号量调整值semadj：子进程不继承父进程的该值，详情请见进程间通信的相关章节。
- 记录锁（fcntl）：子进程不继承父进程的记录锁。比较有意思的地方是文件锁flock子进程是继承的。
- 已用的时间times：子进程将该值初始化成0。

4.9 system函数

前面提到了fork函数、exec系列函数、wait系列函数。库将这些接口糅合在一起，提供了一个system函数。程序可以通过调用system函数，来执行任意的shell命令。相信很多程序员都用过system函数，因为它起到了一个粘合剂的作用，可以让C程序很方便地调用其他语言编写的程序。同时，相信有很多程序员被system函数折磨过，当出现错误时，如何根据system函数的返回值，定位失败的原因是个比较头疼的问题。下面我们来细细展开。

4.9.1 system函数接口

system函数的接口定义如下：

```
#include <stdlib.h>
int system(const char *command);
```

这里将需要执行的命令作为command参数，传给system函数，该函数就帮你执行该命令。这样看来system最大的好处就在于使用方便。不需要自己来调用fork、exec和waitpid，也不需要自己处理错误，处理信号，方便省心。

但是system函数的缺点也是很明显的。首先是效率，使用system运行命令时，一般要创建两个进程，一个是shell进程，另外一个或多个是用于shell所执行的命令。如果对效率要求比较高，最好是自己直接调用fork和exec来执行既定的程序。

从进程的角度来看，调用system的函数，首先会创建一个子进程shell，然后shell会创建子进程来执行command，如图4-15所示。

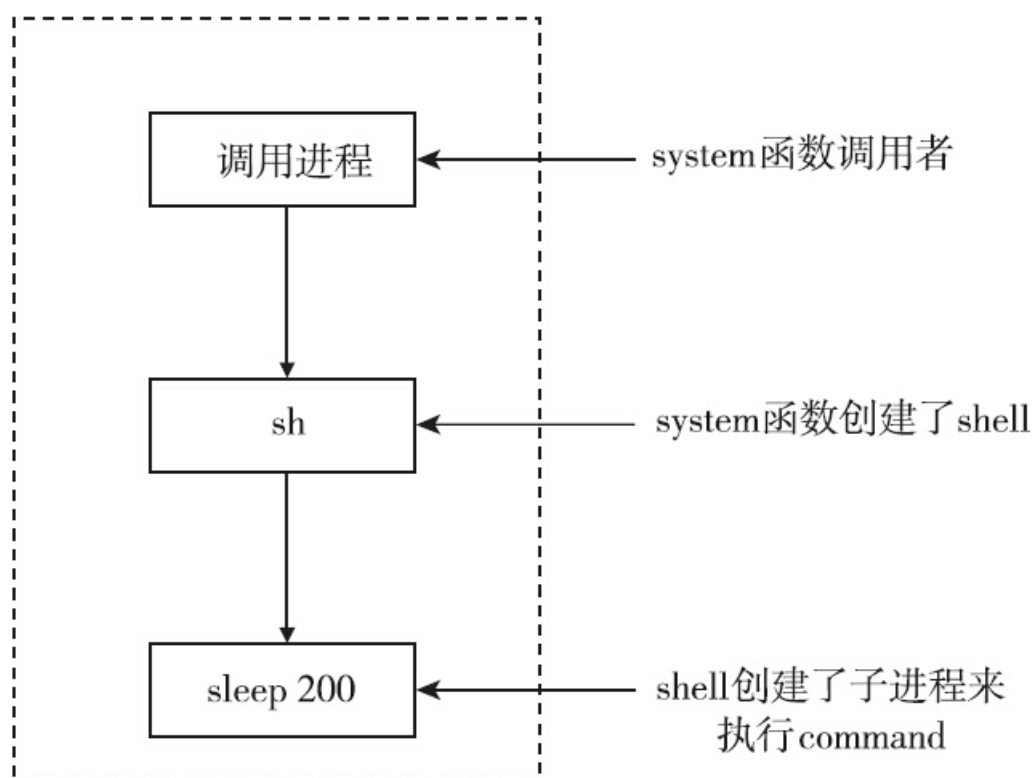


图4-15 system函数的实现

调用system函数后，命令是否运行成功是我们最关心的事情。但是system的返回值比较复杂，下面通过一个简化的不完备（没有处理信号）的system实现来讲述system函数的返回值，代码如下：

```
#include<unistd.h>
```

```

#include<sys/wait.h>
#include<sys/types.h>
int system(char* command)
{
    int status ;
    pid_t child;
    switch(child = fork())
    {
        case -1:
            return -1;
        case 0:
            execl("/bin/sh", "sh", "-c", command, NULL);
            exit(127);
        default:
            while(waitpid(child, &status, 0) < 0)
            {
                /*如果系统调用被中断，则重启系统调用*/
            }
    }
}

```

下面我们来分别讲述system函数的返回值。

(1) 当command为NULL时，返回0或1

正常情况下，不会这样用system。但是command为NULL是有用的，用户可以通过调用system(NULL)来探测shell是否可用。如果shell存在并且可用，则返回1，如果系统里面压根就没有shell，这种情况下，shell就是不可用的，返回0。那么何种情况下shell不可用呢？比如system函数运行在非Unix系统上，再比如程序调用system之前，执行过了chroot，这些情况下shell都可能无法使用。

command为NULL的情况从简化版的代码段中看不出来，但是从glibc的system函数源码中可以看出端倪：

```

glibc-2.17/sysdeps/posix/system.c
-----
int
__libc_system (const char *line)
{
    if (line == NULL)
        return do_system ("exit 0") == 0;
    .....
}
weak_alias (__libc_system, system)

```

(2) 创建进程（fork）失败，或者获取子进程终止状态（waitpid）失败，则返回-1

创建进程失败的情况比较少见，比较容易想到的也就是创建了太多的进程，超出了系统的限制。但是等待子进程终止状态失败，是比较容易造出来的。

前面讲过，子进程退出的时候，如果SIGCHLD的信号处理函数是SIG_IGN或用户设置了SA_NOCLDWAIT标志位，那么子进程就不进入僵尸状态等待父进程wait了，直接自行了断，灰飞烟灭。但是system函数的内部实现会调用waitpid来获取子进程的退出状态。这就是父子之前没有协调好造成的错误。这种情况下，system返回-1，errno为ECHLD。

这种错误的示范代码如下：

```
signal(SIGCHLD,SIG_IGN);/*返回

-1的根源在于此处

*/
if((status = system(command)) < 0)
{
    fprintf(stderr,"system return %d (%s)\n",
    status,strerror(errno));
    return -2;
}
```

这种情况下，总是返回-1，错误码是ECHLD，如下所示：

```
manu@manu-hacks:~$ ./t_sys_err "ls"
system_return.c t_sys_err t_sys_err t_sys_null t_system.c t_system_null.c
system_return -1 (No child processes)
```

所以需要调用system函数的时候，先要确认SIGCHLD是否被设为SIG_IGN。如果是，system就会返回-1，而无法判断command执行成功与否。

（3）如果子进程不能执行shell，那么system返回值会与_exit（127）终止时一样

示例代码如下：

```
case 0:
    execl("/bin/sh","sh","-c",command,NULL);
    _exit(127);
```

这里如果执行execl失败，就会执行到_exit（127），否则不会执行到_exit（127）。

（4）如果所有的系统调用都执行成功，system函数就会返回执行command的子shell的终止状态

因为shell的终止状态是其执行最后一条命令的退出状态。这种情况下就和获取子进程的退出状态一样了。前文详细提到过，可以根据下面的接口来判断：

```
WIFEXITED(status)
WEXITSTATUS(status)
WIFSIGNALED(status)
WTERMSIG(status)
WCOREDUMP(status)
```

综上所述，在`command`不等于NULL的情况下，正确判断`system`返回值的方法如下：

```
if((status = system(command) ) == -1)
{
    fprintf(stderr,"system() function return -1 (%s)\n",
            strerror(errno));
}
else if(WIFEXITED(status) && WEXITSTATUS(status) == 127)
{
    fprintf(stderr,"cannot invoke shell to exec command(%s)\n",command);
}
else
    print_wait_exit(status);
```

其中`print_wait_exit`函数就是前文介绍的通过宏来判断进程的终止状态。

可以测试一下上面的方法。下面的`t_sys`可执行程序是笔者用C写的一个工具，该工具的执行需要1个参数，`argv[1]`用于接受要执行的`command`，这里将用上面提到的方法来判断`command`的执行情况：

```
./t_sys "ls"
system_return.c  t_sys      t_sys_err  t_sys_null  t_system.c  t_system_null.c
status = 0
normal termination,exit status = 0
./t_sys "sleep 100" /*在另一终端向

sleep 进程发送

SIGINT信号

*/
status = 2
abnormal termination,signal number =2
./t_sys "nosuchcmd" /*执行一个不存在的命令

*/
sh: 1: nosuchcmd: not found
cannot invoke shell to exec command(nosuchcmd)
```

4.9.2 system函数与信号

4.9.1节介绍了system函数的用法，并且引入了一个system函数的简单不完备的实现。之所以说是不完备的，是因为没有考虑信号。正确地处理信号，将会给system的实现带来复杂度。

首先要考虑SIGCHLD。如果调用system函数的进程还存在其他子进程，并且对SIGCHLD信号的处理函数也执行了wait（）。那么这种情况下，由system（）创建的子进程退出并产生SIGCHLD信号时，主程序的信号处理函数就可能先被执行，导致system函数内部的waitpid无法等待子进程的退出，这就产生了竞争。这种竞争带来的危害是双方面的：

- 程序会误认为自己调用fork创建的子进程退出了。
- system函数内部的waitpid返回失败，无法获取内部子进程的终止状态。

鉴于上述原因，system运行期间必须要暂时阻塞SIGCHLD信号。

其他需要考虑的信号还有由终端的中断操作（一般是ctrl+c）和退出操作（一般是ctrl+\）产生的SIGINT信号和SIGQUIT信号。

调用system函数会创建shell子进程，然后由shell子进程再创建子进程来执行command。

那么这三个进程又是如何应对的呢？SUSv3标准规定：

- 调用system函数的进程，需要忽略SIGINT和SIGQUIT信号。
- system函数内部创建的进程，要恢复对SIGINT和SIGQUIT的默认处理。

从逻辑上讲，当命令传入给system开始执行时，调用system函数的进程，其实已经放弃了控制权。所以调用system函数的进程不应该响应SIGINT信号和SIGQUIT信号，而应该由system内部创建的子进程来负责响应。考虑到system函数执行的可能是交互式应用，交给system创建的子进程来响应SIGINT和SIGQUIT信号更合情合理。

用更通俗的话来讲，就是调用system函数，在system返回之前会忽略SIGINT和SIGQUIT，无论是调用采用终端的操作（ctrl+c或ctrl+\），还是采用kill来发送SIGINT或SIGQUIT信号，调用system函数的进程都会不动如山。但是system内部创建的执行command的子进程，对SIGINT和SIGQUIT的响应是默认值，也就是说会杀掉响应的子进程而导致system函数的返回。

相对于glibc的system函数实现，《Linux/Unix系统编程手册》提供了一个可读性更好的版本，对实

现感兴趣的朋友，可以参阅该书里面的实现。

可以验证下system对SIGINT及SIGQUIT信号的行为模式是否如前所述。对t_sys对应的进程执行kill-SIGINT，进程t_sys无动于衷。但是在另一终端，对sleep 1000对应的进程发送SIGINT信号，立刻就会出现如下打印：

```
./t_sys "sleep 1000"  
status = 2  
abnormal termination,signal number =2
```

4.10 总结

进程是操作系统非常重要的概念。和程序相比，进程是有生命的，是流动的。本章介绍了进程的一生，从进程被创建到调用`exec`奔向新生活，从进程退出到父进程等待子进程，另外还介绍了上述接口的综合即`system`函数，以及通过`system`函数来执行程序。

第5章 进程控制：状态、调度和优先级

第4章介绍了进程的一生，从创建（fork或vfork）到走向新的征程（exec），从退出（exit或_exit）到被父进程或init进程“收尸”（wait）。

本章将介绍进程的其他方面，主要包括：

- 进程在其或长或短的一生中可能处于的状态。
- 内核如何调度进程使用CPU资源。
- 进程如何调整优先级，以求获得更多或更少的CPU资源。
- 对于有实时性要求的进程如何设置调度策略以满足其要求。
- 如何把进程绑定到某个或某些CPU上执行。

5.1 进程的状态

故飘风不终朝，骤雨不终日。孰为此者？天地。天地尚不能久，而况於人乎？

——老子《道德经》

就像人不可能一刻不停地工作一样，进程也无法始终占有CPU运行。原因有三：

- 进程可能需要等待某种外部条件的满足，在条件满足之前，进程是无法继续执行的。这种情况下，该进程继续占有CPU就是对CPU资源的浪费。

- Linux是多用户多任务的操作系统，可能同时存在多个可以运行的进程，进程个数可能远远多于CPU的个数。一个进程始终占有CPU对其他进程来说是不公平的，进程调度器会在合适的时机，选择合适的进程使用CPU资源。

- Linux进程支持软实时，实时进程的优先级高于普通进程，实时进程之间也有优先级的差别。软实时进程进入可运行状态的时候，可能会发生抢占，抢占当前运行的进程。

下面，首先来讨论一下进程的状态。

5.1.1 进程状态概述

Linux下，进程的状态有以下7种，见表5-1。

表5-1 进程的7种状态

进 程 状 态	说 明
TASK_RUNNING	可运行状态。但未必正在使用 CPU，也许在等待调度
TASK_INTERRUPTIBLE	可中断的睡眠状态。在等待某个条件的完成
TASK_UNINTERRUPTIBLE	不可中断的睡眠状态。与可中断的睡眠状态类似，但是不会被信号中断
TASK_STOPPED	暂停状态。进程收到某信号，运行被停止
TASK_TRACED	被跟踪状态。和暂停状态有些类似，进程被停止，被另一个进程跟踪
EXIT_ZOMBIE	僵尸状态。进程已经退出，但是尚未被父进程或 init 进程“收尸”
EXIT_DEAD	真正死亡的状态。进程停留在该状态的时间极短，很难观察到

1.可运行状态

首先是可运行状态。该状态的名称为TASK_RUNNING，严格来说这个名字是不准确的，因为该状态的确切含义是可运行状态，并非一定是在占有CPU运行，将该状态称为TASK_RUNABLE会更准确。

有人说Linux进程有8种状态，这种说法也是对的。因为TASK_RUNNIING可以根据是否在CPU上运行，进一步细分成RUNNING和READY两种状态（如图5-1所示）。处于READY状态的进程表示，它们随时可以投入运行，只不过由于CPU资源有限，调度器暂时并未选中它运行。

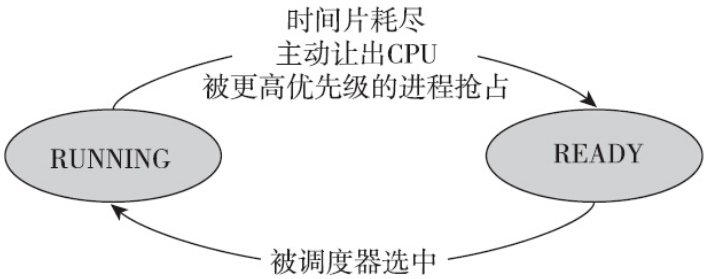


图5-1 READY和RUNNING状态之间的切换

处于可运行状态的进程是进程调度的对象。如果进程并不处于可运行状态，进程调度器就不会选择它投入运行。在Linux中，每一个CPU都有自己的运行队列，事实上还不止一个，根据进程所属调度类别的不同，可运行状态的进程也会位于不同的队列上：如果是实时进程（属于实时调度类），则根据优先级的情况，落在相应的优先级的队列上；如果是普通进程（属于完全公平调度类），则根据虚拟运行时间的大小，落在红黑树的相应位置上。这样进程调度器就可以根据一定的算法从运行队列上挑选合适的进程来使用CPU资源。

处于RUNNING状态的进程，可能正在执行用户态（user-mode）代码，也可能正在执行内核态（kernel-mode）代码，内核提供了进一步的区分和统计。Linux提供的time命令可以统计进程在用户态和内核态消耗的CPU时间：

```
manu@manu-rush:~$ time sleep 2
real    0m2.009s
user    0m0.001s
sys     0m0.002s
```

time命令统计了三种时间：实际时间、用户CPU时间和系统CPU时间。其中实际时间最好理解，就是日常生活中的时间（墙上时间，wall clock time），即进程从开始到终止，一共执行了多久。user一行统计的是进程执行用户态代码消耗的CPU时间；sys一行统计的是进程在内核态运行所消耗的CPU时间。

如何区分用户态CPU时间和内核态CPU时间呢？我们举例来说明。如果进程在执行加减乘除或浮点数计算或排序等操作时，尽管这些操作

正在消耗CPU资源，但是和内核并没有太多的关系，CPU大部分时间都在执行用户态的指令。这种场景下，我们称CPU时间消耗在用户态。如果进程频繁地执行创建进程、销毁进程、分配内存、操作文件等操作，那么进程不得不频繁地陷入内核执行系统调用，这些时间都累加在进程的内核态CPU时间。

对于这三种时间，最容易产生的误解的是 $real\ time = user\ time + sys\ time$ 。这种想法是错误的。在单核系统上， $real\ time$ 总是不小于 $user\ time$ 与 $sys\ time$ 的总和。但是在多核系统上， $user\ time$ 与 $sys\ time$ 的总和可以大于 $real\ time$ 。利用这三个时间，我们可以计算出程序的CPU使用率：

```
cpu_usage = ((user time) + (sys time))/(real time)
```

在多核处理器情况下，`cpu_usage`如果大于1，则表示该进程是计算密集型（CPU bound）的进程，且`cpu_usage`的值越大，表示越充分地利用了多处理器的并行运行优势；如果`cpu_usage`的值小于1，则表示进程为I/O密集型（I/O bound）的进程，多核并行的优势并不明显。

`time`命令的问题在于要等进程运行完毕后，才能获取到进程的统计信息，正所谓盖棺定论。有些时候，我们需要了解正在运行的进程：它运行了多久，内核态CPU时间和用户态CPU时间分别是多少？`procf`s在`/proc/PID/stat`中提供了相关的信息：

```
manu@manu-rush:~$ cat /proc/8283/stat
8283 (stress) R 8282 8282 7015 34817 8282 4218944 35 0 0 0 15988 35

0 0 20 0 1
0 3551036 7405568 24 18446744073709551615 4194304 4213100 140736349760736
140736349760296 139793990053869 0 0 0 0 0 0 17 0 0 0 0 0 6311448 6312216
17915904 140736349767962 140736349767974 140736349767974 140736349769704 0
```

数组中的每个字段都有自己独特的含义。如果从0开始计数，那么字段13对应的是进程消耗的用户态CPU时间，字段14记录的是进程消耗的内核态CPU时间。两者的单位是时钟嘀嗒（clock tick）。

一个时钟嘀嗒是多久？可以通过如下命令来获取：

```
grep CONFIG_HZ /boot/config-`uname -r`
CONFIG_HZ_250=y
CONFIG_HZ=250
```

当配置内核的时候，有100Hz、250Hz、300Hz和1000Hz这4个选项。如果配置的频率为250Hz，那么1秒钟就有250个时钟嘀嗒，即每过4ms，增加一个时钟嘀嗒（内核的`jiffies++`）。

系统提供了`pidstat`命令，通过该命令也可以获取到各个进程的CPU使用情况，如图5-2所示。

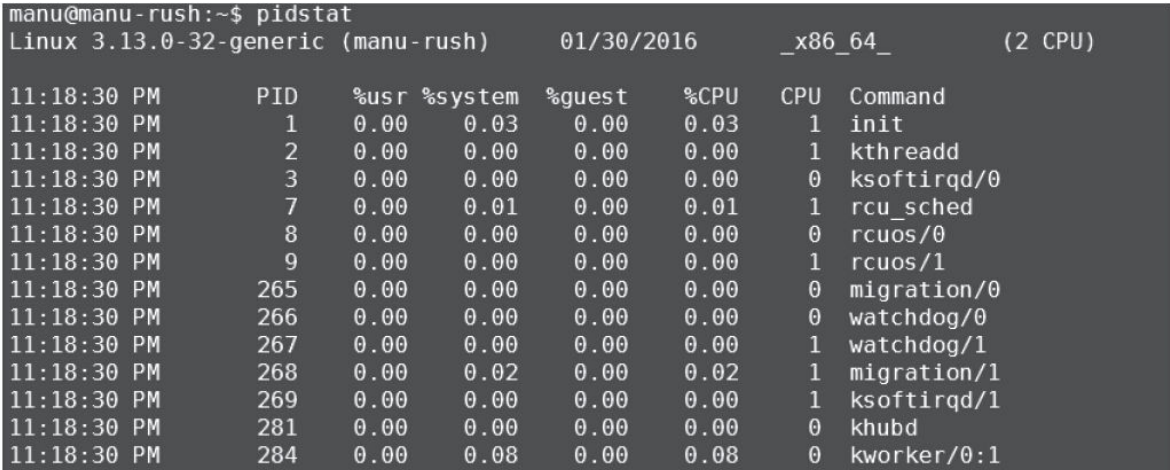


图5-2 使用`pidstat`观察CPU的使用情况

`pidstat`可以通过`-p`参数指定观察的进程，从而可以获取到该进程的CPU使用情况，包括用户态CPU时间和内核态CPU时间，如图5-3所示。

```
manu@manu-rush:~$ pidstat -p 3107 2
```

Linux 3.13.0-32-generic (manu-rush)		01/30/2016		_x86_64_		(2 CPU)	
11:22:01 PM	PID	%usr	%system	%guest	%CPU	CPU	Command
11:22:03 PM	3107	99.00	0.50	0.00	99.50	1	stress
11:22:05 PM	3107	99.00	1.00	0.00	100.00	1	stress
11:22:07 PM	3107	99.00	0.50	0.00	99.50	1	stress
11:22:09 PM	3107	99.50	1.00	0.00	100.50	1	stress

图5-3 使用pidstat观察特定进程的CPU使用情况

如何获得进程的实际运行时间呢？通过ps命令的etime（elapsed time的缩写）可以获取该值：

```
manu@manu-rush:~$ ps -p 8283 -o etime,cmd,pid
```

ELAPSED CMD	PID
02:39 stress -c 1	8283

2.可中断睡眠状态和不可中断睡眠状态

进程并不总是处于可运行的状态。有些进程需要和慢速设备打交道。比如进程和磁盘进行交互，相关的系统调用消耗的时间是非常长的（可能在毫秒数量级甚至会更久），进程需要等待这些操作完成才可以执行接下来的指令。有些进程需要等待某种特定条件（比如进程等待子进程退出、等待socket连接、尝试获得锁、等待信号量等）得到满足后方可执行，而等待的时间往往是不可预估的。在这种情况下，进程依然占用CPU就不合适了，对CPU资源而言，这是一种极大的浪费。因此内核会将该进程的状态改变成其他状态，将其从CPU的运行队列中移除，同时调度器选择其他的进程来使用CPU资源。

Linux存在两种睡眠的状态：可中断的睡眠状态（TASK_INTERRUPTIBLE）和不可中断的睡眠状态（TASK_UNINTERRUPTIBLE）。这两种睡眠状态是很类似的。两者的区别就在于能否响应收到的信号。

处于可中断的睡眠状态的进程，返回到可运行的状态有以下两种可能性：

- 等待的事件发生了，继续运行的条件满足了。
- 收到未被屏蔽的信号。

当处于可中断睡眠状态的进程收到信号时，会返回EINTR给用户空间。程序员需要检测返回值，并做出正确的处理。

但是对于不可中断的睡眠状态，只有一种可能性能使其返回到可运行的状态，即等待的事件发生了，继续运行的条件满足了（如图5-4所示）。



图5-4 可运行状态与休眠状态之间的切换

TASK_UNINTERRUPTIBLE状态存在的意义在于，内核中某些处理流程是不应该被打断的，如果响应异步信号，程序的执行流程中就会插入一段用于处理异步信号的流程，原有的流程就被中断了。因此当进程在对某些硬件进行某些操作时（比如进程调用read系统调用对某个文件进行读操作，read系统调用最终执行对应设备驱动的代码，并与对应的物理设备交互），需要使用TASK_UNINTERRUPTIBLE状态把进程保护起来，以避免进程与设备的交互过程被打断，致使设备陷入不可控的状态。

TASK_UNINTERRUPTIBLE是一种很危险的状态，因为进程进入该状态后，刀枪不入，任何信号都无法打断它。我们无法通过信号杀死一个处于不可中断的休眠状态的进程，SIGKILL信号也不行。

正常情况下，进程处于TASK_UNINTERRUPTIBLE状态的时间会非常短暂，进程不应该长时间处于不可中断的睡眠状态，但是这种情况确

实可能会发生（内核代码流程中可能有bug，或者用户内核模块中的相关机制不合理都会导致某些进程长时间处于D状态）。举例来讲，当通过NFS访问远程目录时，异地文件系统的异常可能会使进程进入该状态。如果远端的文件系统始终异常，使进程的I/O请求得不到满足，该进程会一直处于TASK_UNINTERRUPTIBLE状态，无法杀死，除了重启Linux机器之外，无药可救。

内核提供了hung task检测机制，它会启动一个名为khungtaskd的内核线程来检测处于TASK_UNINTERRUPTIBLE状态的进程是否已经失控。khungtaskd定期被唤醒（默认是120秒），它会遍历所有处于TASK_UNINTERRUPTIBLE状态的进程进行检查，如果某进程超过120秒未获得调度，那么内核就会打印出警告信息和该进程的堆栈信息。

120秒这个时间是可以定制的，内核提供了控制选项：

```
root@manu-rush:~# sysctl kernel.hung_task_timeout_secs
kernel.hung_task_timeout_secs = 120
```

关于khungtaskd的更多细节，可以阅读内核kernel/hung_task.c代码。

无论进程处于可中断的睡眠状态，还是不可中断的睡眠状态，我们都可能会希望了解进程停在什么位置或在等待什么资源。procfs的wchan提供了这方面的信息，wchan是wait channel的含义。ps命令也可以通过wchan获得该信息：

```
manu@manu-rush:~$ echo $$
3828
manu@manu-rush:~$ cat /proc/3828/wchan
do_wait
manu@manu-rush:~$ ps -p 3828 -o pid,wchan,cmd
PID WCHAN CMD
3828 wait -bash
```

另外一种方法是查看进程的stack信息，方法如下所示：

```
manu@manu-rush:~$ sudo cat /proc/3828/stack
[<ffffffff8106d2c4>] do_wait+0x1e4/0x260
[<ffffffff8106e213>] Sys_wait4+0xa3/0x100
[<ffffffff8176847f>] trace_sys+0xe1/0xe6
[<ffffffffffffffff>] 0xffffffffffffffff
```

通过procfs的wchan和stack，不难看出，当前的bash正在等待子进程的退出。

3.睡眠进程和等待队列

进程无论是处于可中断的睡眠状态还是不可中断的睡眠状态，有一个数据结构是绕不开的：等待队列（wait queue）。进程但凡需要休眠，必然是等待某种资源或等待某个事件，内核必须想办法将进程和它等待的资源（或事件）关联起来，当等待的资源可用或等待的事件已发生时，可以及时地唤醒相关的进程。内核采用的方法是等待队列。

等待队列作为Linux内核中的基础数据结构和进程调度紧密地结合在一起。当进程需要等待特定事件时，就将其放置在合适的等待队列上，因此等待队列对应的是一组进入休眠状态的进程，当等待的事件发生时（或者说等待的条件满足时），这组进程会被唤醒，这类事件通常包括：中断（比如DISK I/O完成）、进程同步、休眠时间到时等。

内核使用双向链表来实现等待队列，每个等待队列都可以用等待队列头来标识，等待队列头的定义如下：

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

进程需要休眠的时候，需要定义一个等待队列元素，将该元素挂入合适的等待队列，等待队列元素的定义如下：

```
typedef struct __wait_queue wait_queue_t;
struct __wait_queue {
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE 0x01
    void *private;
    wait_queue_func_t func;
    struct list_head task_list;
};
```

等待队列上的每个等待队列元素，都对应于一个处于睡眠状态的进程（如图5-5所示）。

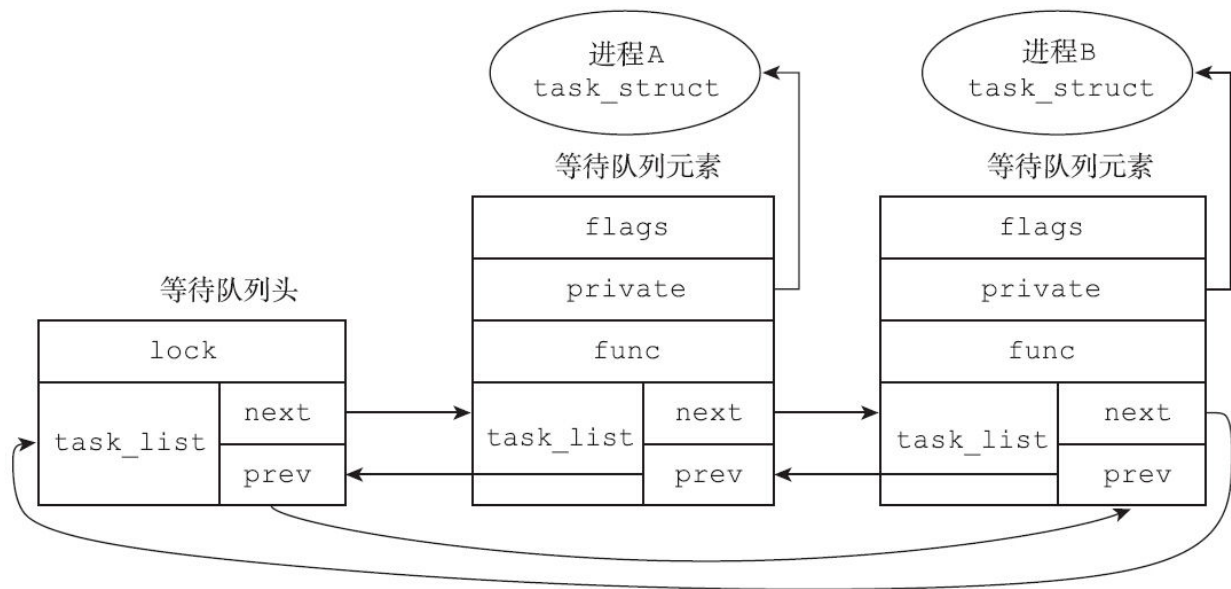


图5-5 睡眠进程与等待队列

内核如何使用等待队列完成睡眠，以及条件满足之后如何唤醒对应的进程呢？

首先要定义和初始化等待队列头部。等待队列头部相当于一杆大旗，没有这杆大旗，将来的等待队列元素将成为“孤魂野鬼”，无处安放。内核提供了init_waitqueue_head和DECLARE_WAIT_QUEUE_HEAD两个宏，用来初始化等待队列头部。

其次，当进程需要睡眠时，需要定义等待队列元素。内核提供了init_waitqueue_entry函数和init_waitqueue_func_entry函数来完成等待队列元素的初始化：

```
static inline void init_waitqueue_entry(wait_queue_t *q, struct task_struct *p)
{
    q->flags = 0;
    q->private = p;
    q->func = default_wake_function; /* 通用的唤醒回调函数 */

}

static inline void init_waitqueue_func_entry(wait_queue_t *q,
                                             wait_queue_func_t func)
{
    q->flags = 0;
    q->private = NULL;
    q->func = func;
}
```

除此以外，内核还提供了宏DECLARE_WAITQUEUE，也可用来初始化等待队列元素：

```
#define __WAITQUEUE_INITIALIZER(name, tsk) { \
    .private = tsk, \
    .func = default_wake_function, \
    .task_list = { NULL, NULL } \
}
#define DECLARE_WAITQUEUE(name, tsk) \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

从等待队列元素的初始化函数或初始化宏不难看出，等待队列元素的private成员变量指向了进程的进程描述符task_struct，因此就有了等待队列元素，就可以将进程挂入对应的等待队列了。

第三步是将等待队列元素（即睡眠进程）放入合适的等待队列中。内核同时提供了add_wait_queue和add_wait_queue_exclusive两个函数来把等待队列元素添加到等待队列头部指向的双向链表，代码如下：

```
void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;
    wait->flags |= WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock, flags);
    __add_wait_queue(q, wait);
    spin_unlock_irqrestore(&q->lock, flags);
}

void add_wait_queue_exclusive(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;
    wait->flags |= WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock, flags);
    __add_wait_queue_tail(q, wait);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

这两个函数的区别在于：

- 一个等待队列元素设置了WQ_FLAG_EXCLUSIVE标志位，而另一个则没有。
- 一个等待队列元素放到了等待队列的尾部，而另一个则放到了等待队列的头部。

同样是添加到等待队列，为何同时提供了两个函数，WQ_FLAG_EXCLUSIVE标志位到底有什么作用？

不妨来思考如下问题：如果存在多个进程在等待同一个条件满足或同一个事件发生（即等待队列上有多个等待队列元素），那么当条件满足时，应该把所有进程一并唤醒还是只唤醒某一个或某几个进程？

答案是具体情况具体分析。有时候需要唤醒等待队列上的所有进程，但又有些时候唤醒操作需要具有排他性（EXCLUSIVE）。比如多个进程等待临界区资源，当锁的持有者释放锁时，如果内核将所有等待在该锁上的进程一起唤醒，那么最终也只能有一个进程竞争到锁资源，而大多数的竞争者，不过是从休眠中醒来，然后继续休眠，这会浪费CPU资源，如果等待队列中的进程数目很大，还会严重影响性能。这就是所谓的惊群效应（thundering herd problem）。因此内核提供了WQ_FLAG_EXCLUSIVE标志位来实现互斥等待，add_wait_queue_exclusive函数会将带有该标志位的等待队列元素添加到等待队列的尾部。当内核唤醒等待队列上的进程时，等待队列元素中的WQ_FLAG_EXCLUSIVE标志位会影响唤醒行为，比如wake_up宏，它唤醒第一个带有WQ_FLAG_EXCLUSIVE标志位的进程后就会停止。

事实上，当内核需要等待某个条件满足而不得不休眠（或是可中断的睡眠，或是不可中断的睡眠）时，内核封装了一些宏来完成前面提到的流程。这些宏包括：

```
wait_event(wq, condition)
wait_event_timeout(wq, condition, timeout)
wait_event_interruptible(wq, condition)
wait_event_interruptible_timeout(wq, condition, timeout)
```

第一个参数指向的是等待队列头部，表示进程会睡眠在该队列上。进程醒来时，condition需要得到满足，否则继续阻塞。其中wait_event和wait_event_interruptible的区别在于，睡眠过程中，前者的进程状态是不可中断的睡眠状态，不能被信号中断，而后者是可中断的睡眠状态，可以被信号中断。名字中带有_timeout的宏意味着阻塞等待的超时时间，以jiffy为单位，当超时时间到达时，无论condition是否满足，均返回。

我们不妨以wait_event宏为例，欣赏一下内核是如何使用等待队列，等待某个条件的满足的：

```
#define wait_event(wq, condition) \
do { \
    if (condition) \
        break; \
    __wait_event(wq, condition); \
} while (0)
#define __wait_event(wq, condition) \
do { \
    DEFINE_WAIT(__wait); \
    for (;;) { \
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \
        if (condition) \
            break; \
        schedule(); \
    } \
    finish_wait(&wq, &__wait); \
} while (0)
void
prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)
{
    unsigned long flags;
    wait->flags &= ~WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock, flags);
    if (list_empty(&wait->task_list))
        __add_wait_queue(q, wait);
    set_current_state(state);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

prepare_to_wait函数负责将等待队列元素添加到对应的等待队列，同时将进程的状态设置成TASK_UNINTERRUPTIBLE，完成prepare_to_wait的工作后，会检查条件是否满足条件，如果条件不满足，则调用schedule（）函数，主动让出CPU使用权，等待被唤醒。

有睡眠就要有唤醒，有wait_event系列的宏，与之对应的，就要有wake_up系列的宏，它们必须成对出现。这一组宏有：

```
wake_up(x)
wake_up_nr(x, nr)
wake_up_all(x)
wake_up_interruptible(x)
wake_up_interruptible_nr(x, nr)
wake_up_interruptible_all(x)
```

这些宏和前面wait_event系列宏的配对使用情况如图5-6所示。

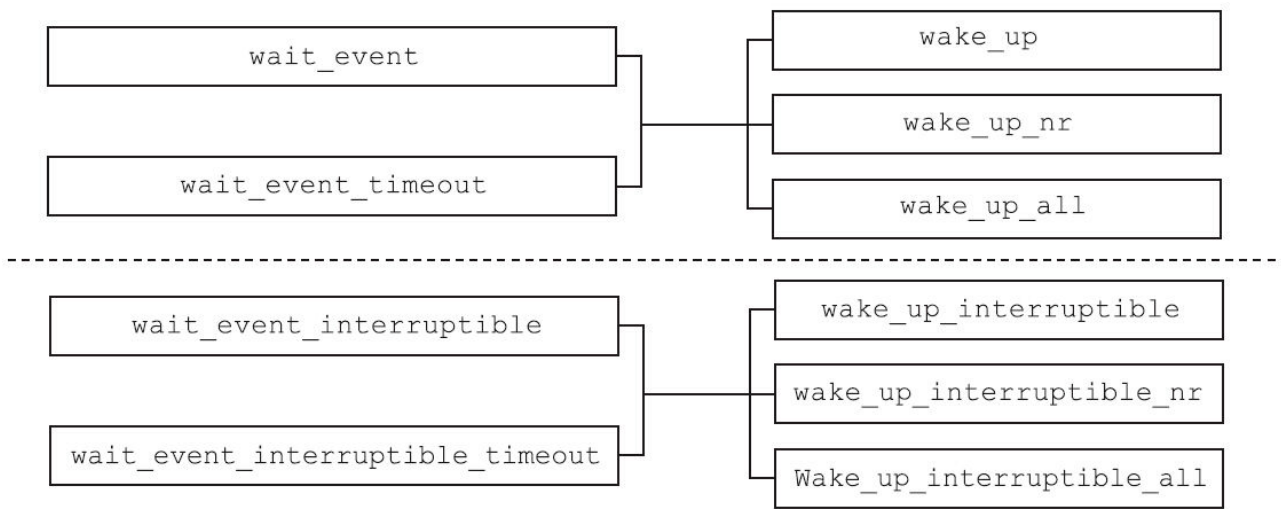


图5-6 wake_event和wake_up配对使用情况

其中该系列宏中，名字里带_interruptible的宏只能唤醒处于TASK_INTERRUPTIBLE状态的进程，而名字中不带_interruptible的宏，既可以唤醒TASK_INTERRUPTIBLE状态的进程，也可以唤醒TASK_UNINTERRUPTIBLE状态的进程。

wake_up系列函数中为什么有些函数后面有_nr和_all这样的后缀？其实不难猜到这些后缀的含义：不带后缀的表示最多只能唤醒一个带有WQ_FLAG_EXCLUSIVE标志位的进程，带_nr的表示可以唤醒nr个带有WQ_FLAG_EXCLUSIVE标志位的进程，而带_all后缀的则表示唤醒等待队列上的所有进程。

这些wake_up系列的宏，其实现部分最终都是通过__wake_up函数的简单封装来实现的，如图5-7所示。

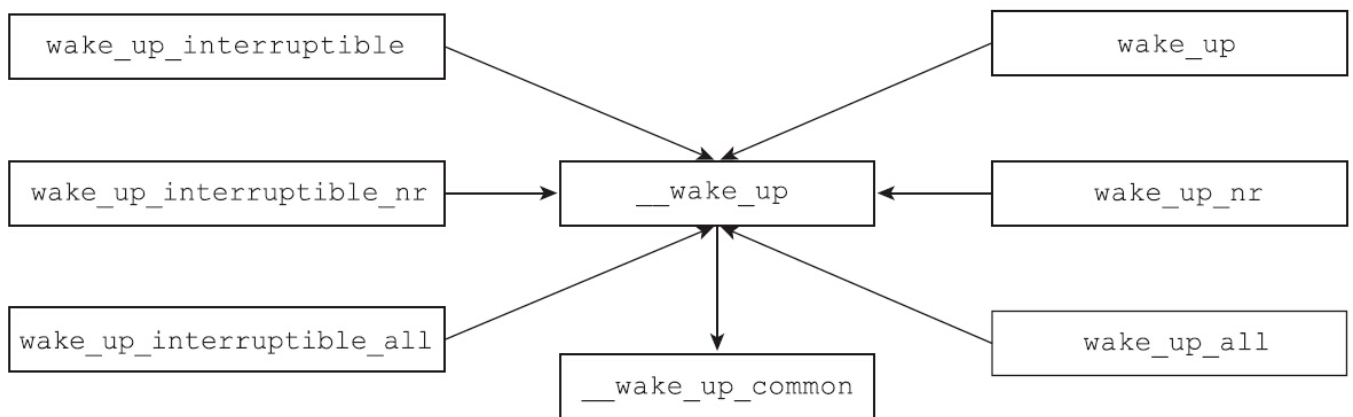


图5-7 wake_up系列函数

下面来分析下__wake_up函数，看看内核是如何唤醒睡眠在等待队列上的进程的，代码如下：

```

void __wake_up(wait_queue_head_t *q, unsigned int mode,
               int nr_exclusive, void *key)
{
    unsigned long flags;
    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, 0, key);
    spin_unlock_irqrestore(&q->lock, flags);
}
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                             int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;
    /*遍历等待队列头部对应的双向链表

*/
    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;
        /*最多唤醒

nr设置了排他性标志位的等待进程，以防止惊群

*/
        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}

```



```
}  
}
```

注意，遍历等待队列上的所有等待队列元素时，对于每一个需要唤醒的进程，执行的是等待队列元素中定义的func，最多唤醒nr_exclusive个带有WQ_FLAG_EXCLUSIVE的等待队列元素。

在初始化等待队列元素的时候，需要注册回调函数func。当内核唤醒该进程时，就会执行等待队列元素中的回调函数。

等待队列元素最常用的回调函数是default_wake_function，就像它的名字一样，是默认的唤醒回调函数。无论是DECLARE_WAITQUEUE还是init_waitqueue_entry，都将等待队列元素的func指向default_wake_function。而default_wake_function仅仅是大名鼎鼎的try_to_wake_up函数的简单封装，代码如下：

```
int default_wake_function(wait_queue_t *curr, unsigned mode, int wake_flags,  
                          void *key)  
{  
    return try_to_wake_up(curr->private, mode, wake_flags);  
}
```

try_to_wake_up是进程调度里非常重要的一个函数，它负责将睡眠的进程唤醒，并将醒来的进程放置到CPU的运行队列中，然后并设置进程的状态为TASK_RUNNING。在本章的后面会对该函数进行详细的分析。

4.TASK_KILLABLE状态

很多文章在介绍TASK_UNINTERRUPTIBLE状态时，都喜欢通过下面的例子来创建一个处于TASK_UNINTERRUPTIBLE状态的进程：

```
#include<stdio.h>  
int main()  
{if(!vfork())  
{  
    sleep(100);  
    printf("hello \n");  
}  
}
```

运行上述代码编出的程序：

```
root@manu-rush:~# ps ax |grep state_d  
5880 pts/2    D+      0:00 ./state_d  
5881 pts/2    S+      0:00 ./state_d
```

很多文章认为，调用vfork函数创建子进程时，子进程在调用exec函数或退出之前，父进程始终处于TASK_UNINTERRUPTIBLE的状态。其实这种说法是错误的。因为很明显，父进程可以轻易地被信号杀死，这证明父进程并不是处于TASK_UNINTERRUPTIBLE的状态。

```
root@manu-hacks:~# ps ax |grep state_d |grep -v grep  
6787 pts/2    D+      0:00 ./state_d  
6788 pts/2    S+      0:00 ./state_d  
root@manu-hacks:~# kill -9 6787  
root@manu-hacks:~# ps ax |grep state_d |grep -v grep  
6788 pts/2    S       0:00 ./state_d而在程序运行的终端
```

```
manu@manu-hacks:~/code/me/c$ ./state_d  
Killed
```

为什么进程的状态显示的是D+，按照ps命令的说法应该是处于不可中断的睡眠状态，可为什么仍然会被信号杀死呢？这好像和前面的讲述并不一致。

事实上，ps命令输出的D状态不能简单地理解成UNINTERRUPTIBLE状态。内核自2.6.25版本起引入了一种新的状态即TASK_KILLABLE状态^[1]。可中断的睡眠状态太容易被信号打断，与之对应，不可中断的睡眠状态完全不可以被信号打断，又容易失控，两者都失之极端。而内核新引入的TASK_KILLABLE状态则介于两者之间，是一种调和状态。该状态行为上类似于TASK_UNINTERRUPTIBLE状态，但是进程收到致命信号（即杀死一个进程的信号）时，进程会被唤醒。

上面的例子中vfork创建子进程之后，ps显示父进程处于D的状态，却依然可以被杀死的原因就是进程并不是处于不可中断的睡眠状态，而是处于TASK_KILLABLE状态。而这种状态，是可以响应致命信号的。

有了该状态，wait_event系列宏也增加了killable的变体，即wait_event_killable宏。该宏会将进程置为TASK_KILLABLE状态，同时睡眠在等待队列上。致命信号SIGKILL可以将其唤醒。

5.TASK_STOPPED状态和TASK_TRACED状态

TASK_STOPPED状态是一种比较特殊的状态。在第4章曾经提到过，SIGSTOP、SIGTSTP、SIGTTIN和SIGTTOU等信号会将进程暂时停止，停止后进程就会进入到该状态。上述4种信号中的SIGSTOP具有和SIGKILL类似的属性，即不能忽略，不能安装新的信号处理函数，不能屏蔽等。当处于TASK_STOPPED状态的进程收到SIGCONT信号后，可以恢复进程的执行（如图5-8所示）。

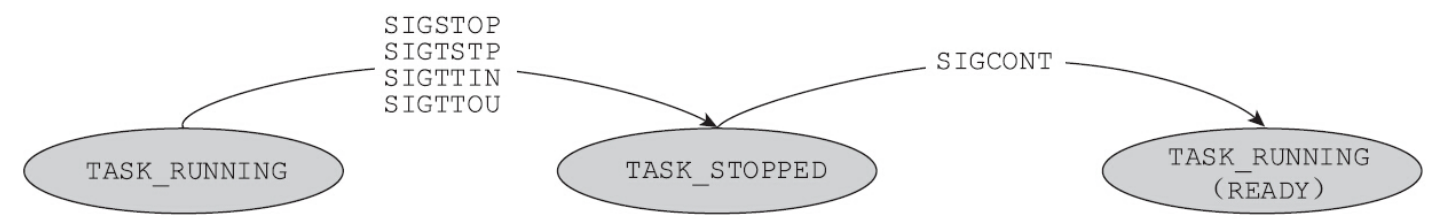


图5-8 可运行状态和暂停状态的切换

TASK_TRACED是被跟踪的状态，进程会停下来等待跟踪它的进程对它进行进一步的操作。如何才能制造出处于TASK_TRACED状态的进程呢？最简单的例子是用gdb调试程序，当进程在断点处停下来时，此时进程处于该状态。

下面用一个最简单的hello程序来验证gdb停下的程序的确处于TASK_TRACED的状态。

在一个终端，gdb将程序停下，停在断点处：

```
Breakpoint 1, main () at hello.c:6
6      printf("hello world\n");
```

在另一个终端查看进程的状态：

```
manu@manu-hacks:~$ ps ax |grep hello
3768 pts/2 S+ 0:00 gdb ./hello
3770 pts/2 t 0:00 /home/manu/code/me/c/hello
manu@manu-hacks:~$ cat /proc/3770/status
Name: hello
State: t (tracing stop)
```

TASK_TRACED和TASK_STOPPED状态的类似之处是都处于暂停状态，不同之处是TASK_TRACED不会被SIGCONT信号唤醒。只有调试进程通过ptrace系统调用，下达PTRACE_CONT、PTRACE_DETACH等指令，或者调试进程退出，被调试的进程才能恢复TASK_RUNNING的状态。

6.EXIT_ZOMBIE状态和EXIT_DEAD状态

EXIT_ZOMBIE和EXIT_DEAD是两种退出状态，严格说来，它们并不是运行状态。当进程处于这两种状态中的任何一种时，它其实已经死去了。内核会将这两种状态记录在进程描述符的exit_state中，不过不想细分的话，可以笼统地说进程处于TASK_DEAD状态。

两种状态的区别在于，如果父进程没有将SIGCHLD信号的处理函数重设为SIG_IGN，或者没有为SIGCHLD设置SA_NOCLDWAIT标志位，那么子进程退出后，会进入僵尸状态等待父进程或init进程来收尸，否则直接进入EXIT_DEAD。如果不停留在僵尸状态，进程的退出是非常快的，因此很难观察到一个进程是否处于EXIT_DEAD状态。

5.1.2 观察进程状态

5.1.1节介绍了进程的状态，本节将介绍如何观察进程当前所处的状态。

在proc文件系统中，在/proc/PID/status中，记录了PID对应进程的状态信息。其中State项记录了该进程的瞬时状态。因为进程状态是不断迁移变化的，所以读出来的结果是瞬时的值。

```
manu@manu-rush:~$ cat /proc/1/status
Name:    init
State:   S (sleeping)
```

procfs中，进程的状态有几种可能的值呢？一起去查看内核的源码。在fs/proc/array.c中，定义了所有可能的值，定义如下：

```
static const char * const task_state_array[] = {
    "R (running)", /* 0 */
    "S (sleeping)", /* 1 */
    "D (disk sleep)", /* 2 */
    "T (stopped)", /* 4 */
    "t (tracing stop)", /* 8 */
    "Z (zombie)", /* 16 */
    "X (dead)", /* 32 */
    "x (dead)", /* 64 */
    "K (wakekill)", /* 128 */
    "W (waking)", /* 256 */
};
```

这几种状态都会从procfs中出现吗？并非如此。

```
static inline const char *get_task_state(struct task_struct *tsk)
{
    unsigned int state = (tsk->state & TASK_REPORT) | tsk->exit_state;
    const char * const *p = &task_state_array[0];
    BUILD_BUG_ON(1 + ilog2(TASK_STATE_MAX) != ARRAY_SIZE(task_state_array));
    while (state) {
        p++;
        state >>= 1;
    }
    return *p;
}
```

只有在TASK_REPORT宏出现的状态加上两个退出状态时，才能出现在procfs中：

```
#define TASK_REPORT (TASK_RUNNING | TASK_INTERRUPTIBLE | \
    __TASK_TRACED) TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
```

从TASK_REPORT宏中可以看出，并没有TASK_DEAD、TASK_WAKEKILL和TASK_WAKING，也就是说在procfs中，无法观察到下面这三个值，它们从不出现：

```
"x (dead)", /* 64 */
"K (wakekill)", /* 128 */
"W (waking)", /* 256 */
```

在vfork那个例子中，在procfs中查询进程状态时，父进程处于D（disk sleep）状态，而并没有出现K（wakekill），原因就在于此。

那么是时候记住，会在procfs中出现的进程状态了：

```
"R (running)",
"S (sleeping)",
"D (disk sleep)",
"T (stopped)",
"t (tracing stop)",
"Z (zombie)",
"X (dead)",
```

这就是传统的进程7状态，如表5-2所示。

表5-2 procfs中的进程状态

procfs 中的值	进程状态
R (running)	TASK_RUNNING
S (sleeping)	TASK_INTERRUPTIBLE
D (disk sleep)	TASK_UNINTERRUPTIBLE

(续)

procfs 中的值	进程状态
T (stopped)	TASK_STOPPED [⊖]
t (tracing stop)	TASK_TRACED [⊖]
Z (zombie)	EXIT_ZOMBIE
X (dead)	EXIT_DEAD

(注：在此处，TASK_STOPPED应为__TASK_STOPPED，为了防止产生不必要的困扰，不做严格区分。)

(注：在此处，TASK_TRACED应为__TASK_TRACED，为了防止产生不必要的困扰，不做严格区分。)

5.2 进程调度概述

进程调度，是任何一个现代操作系统都要解决的问题，它是操作系统相当重要的一个组成部分。首先需要理解的一点是，进程调度器是对处于可运行（TASK_RUNNING）状态的进程进行调度，如果进程并非TASK_RUNNING的状态，那么该进程和进程调度是没有关系的。

Linux是多任务的操作系统，所谓多任务是指系统能够同时并发地执行多个进程，哪怕是单处理器系统。在单处理器系统上支持多任务，会给用户多个进程同时跑的幻觉，事实上多个进程仅仅是轮流使用CPU资源。只有在多处理器系统中，多个进程才能真正地做到同时、并行地执行。

多任务系统可以根据是否支持抢占分成两类：非抢占式多任务和抢占式多任务。在非抢占式多任务的系统中，下一个任务被调度的前提是当前进程主动让出CPU的使用权，因此非抢占式多任务又称为合作型多任务。而抢占式多任务由操作系统来决定进程调度，在某些时间点上，操作系统可以将正在运行的进程调度出去，选择其他进程来执行。毫无疑问，Linux属于抢占式多任务系统。事实上，大多数的现代操作系统都是抢占式的多任务系统。

CPU是一种关键的系统资源。在普通PC上CPU的核数不过4核、8核等，在服务器上可能有16核、32核甚至更多。在系统负载始终比较轻（即可运行状态的进程不多）的情况下，进程调度的重要性并不大。但是如果系统的负载很高，有几百上千的进程处于可运行的状态，那么一套合理高效的调度算法就非常重要了。

此外，不同的进程之间，其行为模式可能存在着巨大的差异。进程的行为模式可以粗略地分成两类：CPU消耗型（CPU bound）和I/O消耗型（I/O bound）。所谓CPU消耗型是指进程因为没有太多的I/O需求，始终处于可运行的状态，始终在执行指令。而I/O消耗型是指进程会有大量I/O请求（比如等待键盘键入、读写块设备上的文件、等待网络I/O等），它处于可执行状态的时间不多，而是将更多的时间耗费在等待上。当然这种划分方法并非绝对的，可能有些进程某段时间表现出CPU消耗型的特征，另一段时间又表现出I/O消耗型的特征。

还有另外一种进程分类的方法，如下。

·交互型进程：这种类型的进程有很多的人机交互，进程会不断地陷入休眠状态，等待键盘和鼠标的输入。但是这种进程对系统的响应时间要求非常高，用户输入之后，进程必须被及时唤醒，否则用户就会觉得系统反应迟钝。比较典型的例子是文本编辑程序和图形处理程序等。

·批处理型进程：这类进程和交互型的进程相反，它不需要和用户交互，通常在后台执行。这样的进程不需要及时的响应。比较典型的例子是编译、大规模科学计算等，一般来说，这种进程总是“被侮辱的和被损害的”。

·实时进程：这类进程优先级比较高，不应该被普通进程和优先级比它低的进程阻塞。一般需要比较短的响应时间。

系统之中，有很多性格各异的进程，这就增加了设计调度器的难度。有一个很有意思的比喻来描述调度器的困境^[1]：Linux内核调度器就像处境尴尬的主妇，满足孩子对晚餐的要求便有可能会伤害到老人的食欲，做出一桌让男女老少都满意的饭菜实在是太难了。

设计一个优秀的进程调度器绝不是一件容易的事情，它还有很多事情需要考虑，很多目标需要达成：

·公平：每一个进程都可以获得调度的机会，不能出现“饿死”的现象。

·良好的调度延迟：尽量确保进程在一定的时间内，总能够获得调度的机会。

·差异化：允许重要的进程获得更多的执行时间。

·支持软实时进程：软实时进程，比普通进程具有更高的优先级。

- 负载均衡：多个CPU之间的负载要均衡，不能出现一些CPU很忙，而另一些CPU很闲的情况。
- 高吞吐量：单位时间内完成的进程个数尽可能多。
- 简单高效：调度算法要高效。不应该在调度上花费太长的时间。
- 低功耗量：在系统并不繁忙的情况下，降低系统的耗电量。

在对称多处理器（SMP）的系统上，存在着多个处理器，那么所有处于可运行状态的进程是应该位于一个队列上，还是每个处理器都要有自己的队列？这大概是进程调度首先要解决的问题。

目前Linux采用的是每个CPU都要有自己的运行队列，即per cpu run queue。每个CPU去自己的运行队列中选择进程，这样就降低了竞争。这种方案还有另外一个好处：缓存重利用。某个进程位于这个CPU的运行队列上，经过多次调度之后，内核趋于选择相同的CPU执行该进程。这种情况下上次运行的变量很可能仍然在CPU的缓存中，这样就提升了效率。

所有的CPU共用一个运行队列这种方案的弊端是显而易见的，尤其是在CPU数目很多的情况下。我们可以想象一下如果存在1024个CPU，都要去同一个运行队列取下一个调度的进程，这种竞争无疑会降低调度器的性能。

但是凡事无绝对，没有最好的，只有最适合的。对于CPU核数比较少的桌面应用来说，只有一个运行队列的Brain Fuck Scheduler（脑残调度器）却表现的异常出色。[\[2\]](#)[\[3\]](#)

Linux选择了每一个CPU都有自己的运行队列这种解决方案。这种选择也带来了一种风险：CPU之间负载不均衡，可能出现一些CPU闲着而另外一些CPU忙不过来的情况。为了解决这个问题，load_balance就闪亮登场了。load_balance的任务就是在一定的时机下，通过将任务从一个CPU的运行队列迁移到另一个CPU的运行队列，来保持CPU之间的负载均衡。

进程调度具体要做哪些事情呢？概括地说，进程调度的职责是挑选下一个执行的进程，如果下一个被调度到的进程和调度前运行的进程不是同一个，则执行上下文切换，将新选择的进程投入运行。

下面根据调度的入口点函数schedule（）来看下进程调度做了哪些事情，代码如下：

```
asmlinkage void __sched schedule(void)
{
    struct task_struct *tsk = current;
    sched_submit_work(tsk);
    __schedule();
}
static void __sched __schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;
    need_resched:
    preempt_disable();
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    rcu_note_context_switch(cpu);
    prev = rq->curr;
    schedule_debug(prev);
    if (sched_feat(HRTICK))
        hrtick_clear(rq);
    raw_spin_lock_irq(&rq->lock);
    switch_count = &prev->nivcsw;
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        if (unlikely(signal_pending_state(prev->state, prev))) {
            prev->state = TASK_RUNNING;
        } else {
            /*先前的进程不再处于可执行状态，需要将其从运行队列中移除出去*/
        }
    }
    deactivate_task(rq, prev, DEQUEUE_SLEEP);
    prev->on_rq = 0;
    if (prev->flags & PF_WQ_WORKER) {
        struct task_struct *to_wakeup;
        to_wakeup = wq_worker_sleeping(prev, cpu);
        if (to_wakeup)
            try_to_wake_up_local(to_wakeup);
    }
    switch_count = &prev->nvcsw;
}
/*调度之前的准备工作*/

*/
```

```
pre_schedule(rq, prev); /*当前

CPU运行队列上没有可运行的进程了，太闲了，需要负载均衡

*/
if (unlikely(!rq->nr_running))
    idle_balance(cpu, rq);
/*将被抢占的进程放入指定的合适的位置

*/
put_prev_task(rq, prev);
/*挑选下一个执行的进程

*/
next = pick_next_task(rq);
/*清除被抢占进程的需要调度的标志位

*/
clear_tsk_need_resched(prev);
rq->skip_clock_update = 0;
/*如果选中的进程与原进程不是同一个进程，则需要上下文切换

*/
if (likely(prev != next)) {
    rq->nr_switches++;
    rq->curr = next;
    ++*switch_count;
    /*上下文切换，切换之后，新选中的进程投入执行

*/
    context_switch(rq, prev, next);
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
} else
    raw_spin_unlock_irq(&rq->lock);
post_schedule(rq);
preempt_enable_no_resched();
if (need_resched())
    goto need_resched;
}
```

Linux是可抢占式内核（Preemptive Kernel），从内核2.6版本开始，Linux不仅支持用户态抢占，也开始支持内核态抢占。可抢占式内核的优势在于可以保证系统的响应时间。当高优先级的任务一旦就绪，总能及时得到CPU的控制权。但是很明显，内核抢占不能随意发生，某些情况下是不允许发生内核抢占的。因此为了更好地支持内核抢占，内核为每一个进程的thread_info引入了preempt_count计数器，数值为0时表示可以抢占，当该计数器的值不为0时，表示禁止抢占。

并不是所有的时机都允许发生内核抢占。以自旋锁为例，在内核可抢占的系统中，自旋锁持有期间不允许发生内核抢占，否则可能会导致其他CPU长期不能获得锁而死等。因此在spin_lock函数中（通过__raw_spin_lock），会调用preempt_disable宏，而该宏会将进程preempt_count计数器的值加1，表示不允许抢占。同样的道理，解锁的时候，会将preempt_count的值减1（通过preempt_enable宏）。

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENTED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

preempt_count的Bit 28是一个很重要的标志位，即PREEMPT_ACTIVE。该标志位用来标记是否正在进行内核抢占。很明显，设置了该标志位之后，preempt_count就不再为0了，因此也就不允许再次发生内核抢占，从而使得正在执行抢占工作的代码不会再次被抢占。

内核的preempt_schedule函数是内核抢占时呼叫调度器的入口，它会调用__schedule函数发起调度。在调用__schedule函数之前，会设置进程的PREEMPT_ACTIVE标志位，表示这是从抢占过程中进入__schedule函数的。

```
asmlinkage void __sched notrace preempt_schedule(void)
{
    struct thread_info *ti = current_thread_info();
    if (likely(ti->preempt_count || irqs_disabled()))
        return;
    do {
        add_preempt_count_notrace(PREEMPT_ACTIVE);
```

```
        _schedule();
        sub_preempt_count_notrace(PREEMPT_ACTIVE);
        barrier();
    } while (need_resched());
}
```

在__schedule函数中，内核会检查进程的PREEMPT_ACTIVE标志位，如果发现了该标志位置位，就不会调用deactivate_task函数将其从运行队列中移除。

PREEMPT_ACTIVE标志位有一个非常重要的作用，即防止不处于TASK_RUNNING状态的进程被抢占过程错误地从运行队列中移除。这句话非常地绕，我们结合__schedule函数的对应代码来分析该标志位的作用。

```
if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
    if (unlikely(signal_pending_state(prev->state, prev))) {
        prev->state = TASK_RUNNING;
    } else {
        deactivate_task(rq, prev, DEQUEUE_SLEEP);
        ...
    }
    ...
}
```

如果进程设置了PREEMPT_ACTIVE标志位，上述代码最外层的条件就不会得到满足。这么做的用意是：如果进程是被抢占而进入了schedule函数，那么即使它不处于TASK_RUNNING状态，也不能把它从运行队列中移除。

为什么这么做？从运行队列中移除不处于TASK_RUNNING状态的进程是schedule函数份内之事，为什么设置了PREEMPT_ACTIVE标志位就不能移除呢？

原因是进程从TASK_RUNNING变成其他状态，是一个过程，在这个过程中可能发生抢占。试想如下场景：一个进程刚把自己设置成TASK_INTERRUPTIBLE，它就被抢占了。因为这时候它还没来得及调用schedule（）主动交出CPU控制权，仍然在CPU上执行，这就是非TASK_RUNNING状态的进程也会被抢占的场景。对于这种场景，抢占流程不应擅自将其从运行队列中移除，因为它的切换过程并未完成。

下面的代码在wait_event系列宏中不断出现，我们以它为例分析上面提到的问题：

```
for (;;) {
    prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
    if (condition)
        break;
    schedule();
}
```

执行完prepare_to_wait语句，本来是要检查条件是否满足的，如果这时候被抢占，假如没有PREEMPT_ACTIVE标志位，那么抢占过程中调用的__schedule函数就会将进程从运行队列中移除。如果本来condition条件满足了，那就错过了唤醒的机会，也许就会永远休眠了。正确的做法是，继续保留在运行队列中，后面还有机会被调度到继续运行，恢复运行后继续判断条件是否满足。

上面讨论了抢占的情况，如果进程不处于TASK_RUNNING的状态，并且PREEMPT_ACTIVE并没有置位，那么就有可能调用deactivate_task函数将其从运行队列中移除。这里说可能是因为，该进程可能存在尚未处理的信号，如果是这种情况它并不会被移除出运行队列，相反会被再次设置成TASK_RUNNING的状态，获得再次被调度到的机会。

__schedule函数的基本流程如图5-9所示。流程图中带有背景色的部分都是调度框架里的hook点。内核的进程调度是模块化的，实现一个新的调度算法，只需要实现一组框架需要的钩子函数即可，内核将会在合适的时机调用这些函数。

不妨以deactivate_task为例，来看下调度框架与具体调度算法中的函数之间的关系。deactivate_task函数的职责可以顾名思义，即进程不再处于TASK_RUNNING的状态，需要将其从对应的运行队列中移除。因此其实现为：

```
static void deactivate_task(struct rq *rq, struct task_struct *p, int flags)
{
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible++;
    dequeue_task(rq, p, flags);
}
static void dequeue_task(struct rq *rq, struct task_struct *p, int flags)
{
    update_rq_clock(rq);
    sched_info_dequeued(p);
    p->sched_class->dequeue_task(rq, p, flags);
}
```

内核会调用进程所属调度类的`dequeue_task`函数，至于调度类的`dequeue_task`函数具体做了哪些事情，完全由具体的调度类来决定。

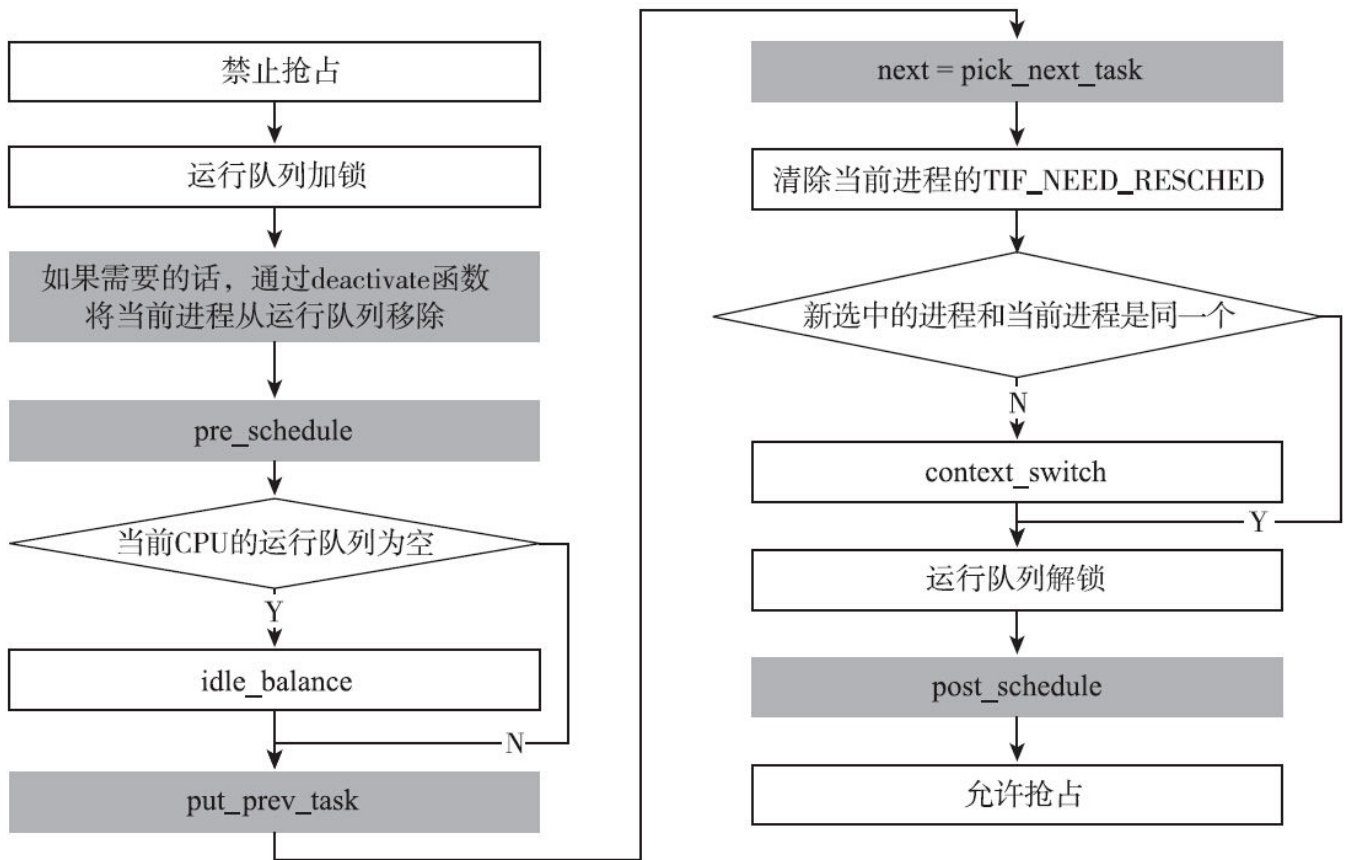


图5-9 schedule函数的基本流程

调用`schedule`函数时，当前进程可能仍然处于可运行的状态（主动让出CPU或被其他进程抢占），因此选择下一个占用CPU的进程之前，需要调用`put_prev_task`函数。该函数的目的是，当前进程被调度出去之前，留给具体调度算法一个时机来更新内部的状态（如图5-9所示）。和`deactivate_task`函数一样，根据当前进程所属的调度类，调用具体的`put_prev_task`函数。

```

static void put_prev_task(struct rq *rq, struct task_struct *prev)
{
    if (prev->on_rq || rq->skip_clock_update < 0)
        update_rq_clock(rq);
    prev->sched_class->put_prev_task(rq, prev);
}
    
```

Linux内核实现了如下4种调度类：

- `stop_sched_class`：停止类
- `rt_sched_class`：实时类
- `fair_sched_class`：完全公平调度类
- `idle_sched_class`：空闲类

这4种调度类是按照优先级顺序排列的，停止类（`stop_sched_class`）具有最高的调度优先级，与之对应的，空闲类（`idle_sched_class`）具有最低的调度优先级。进程调度器挑选下一个执行的进程时，会首先从停止类中挑选进程，如果停止类中没有挑选到可运行的进程，再从实时类中挑选进程，依此类推。

`pick_next_task`函数负责挑选下一个运行的进程，从其实现逻辑中可以看出，系统是按照优先级顺序从调度类中挑选进程的（如图

5-10所示）。

```
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
    const struct sched_class *class;
    struct task_struct *p;
    /*此处是优化，若所有任务都属于公平类，则直接从公平类中挑选下一个类

    */
    if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
        p = fair_sched_class.pick_next_task(rq);
        if (likely(p))
            return p;
    }
    /*按照调度类的优先级，从高到低挑选下一个进程，直到挑选到为止

    */
    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }
    ...
}
```

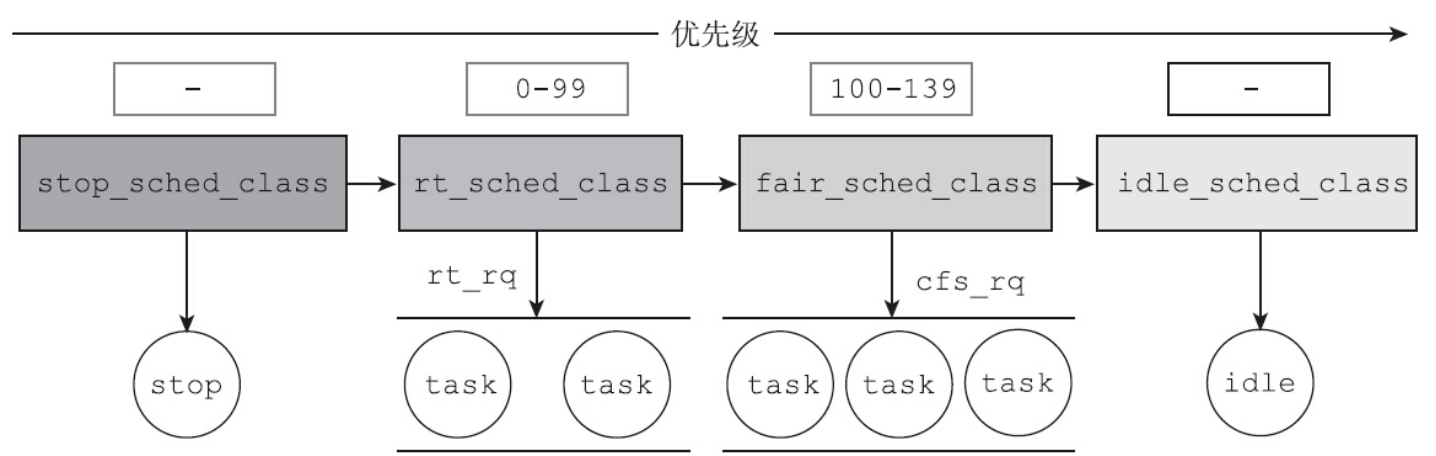


图5-10 进程调度类优先级次序

优先级最高的停止类进程，主要用于多个CPU之间的负载均衡和CPU的热插拔，它所做的事情就是停止正在运行的CPU，以进行任务的迁移或插拔CPU。优先级最低的空闲类，负责将CPU置于停机状态，直到有中断将其唤醒。idle_sched_class类的空闲任务只有在没有其他任务的时候才能被执行。

每一个CPU只有一个停止任务和一个空闲任务。从上面的职责描述也可以看出，这两种调度类属于诸神之战，和应用层的关系并不大。应用层无法将进程设置成停止类进程或空闲类进程。

和应用层关系比较密切的两种调度类是实时类和完全公平调度类，尤其是完全公平调度类。

[1] 参考资料见：刘明Linux调度器BFS简介BFS vs CFS <https://www.ibm.com/developerworks/cn/linux/l-cn-bfs/>。

[2] BFS简介，Linux桌面的极速未来？

[3] BFS vs CFS-Scheduler Comparision: http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockelschulte.pdf。

5.3 普通进程的优先级

本节将停留在进程调度版图中的完全公平调度类（Completely Fair Scheduler，简称CFS）上。事实上，除非将Linux用在特定的领域，否则在大部分时间里所有可运行的进程都属于完全公平调度类。从内核代码pick_next_task函数（该函数负责挑选下一个进程放到CPU上执行）中所做的优化可见一斑。

Linux是多任务系统，在存在多个可运行进程的情况下，系统不能放任当前进程始终占着CPU。每个进程运行多长时间，是任何一个调度算法都不能回避的问题。传统的调度算法面临着一种困境，那就是时间片到底多大才合适？如果时间片太大，进程执行前需要等待的时间就会变长，当CPU运行队列上可运行进程的个数比较多时候尤为明显，用户可能会感觉到明显的延迟。如果时间片太短，进程调度的频率就会增加，考虑到上下文切换也需要花费时间，可以想见，大量的时间都浪费到了进程调度上。

完全公平调度，使用了一种动态时间片的算法。它给每个进程分配了使用CPU的时间比例。进程调度设计上，有一个很重要的指标是调度延迟，即保证每一个可运行的进程都至少运行一次的时间间隔。比如调度延迟是20毫秒，如果运行队列上只有2个同等优先级的进程，那么可以允许每个进程执行10毫秒，如果运行队列上是4个同等优先级的进程，那么，每个进程可以运行5毫秒。

如果可运行的进程比较少，采用这种算法则没有问题。可是如果运行队列上有200个同等优先级的进程怎么办？每个进程运行0.1毫秒？这不是个好主意。因为时间片太小，进程调度过于频繁，上下文切换的开销就不能忽视了。

为了应对这种情况，完全公平调度提供了另一种控制方法：调度最小粒度。调度最小粒度指的是任一进程所运行的时间长度的基准值。任何一个进程，只要分配到了CPU资源，都至少会执行调度最小粒度的时间，除非进程在执行过程中执行了阻塞型的系统调用或主动让出CPU资源（通过sched_yield调用）。

在Linux操作系统中，调度延迟被称为sysctl_sched_latency，记录在/proc/sys/kernel/sched_latency_ns中，而调度最小粒度被称为sysctl_sched_min_granularity，记录在/proc/sys/kernel/sched_min_granularity_ns中，两者的单位都是纳秒。

```
cat /proc/sys/kernel/sched_latency_ns
12000000
cat /proc/sys/kernel/sched_min_granularity_ns
1500000
```

调度延迟和调度最小粒度综合起来看是比较有意思的，它反映了在调度延迟内允许的最大活动进程数目。这个值被称为sched_nr_latency。如果运行队列上可运行状态的进程太多，超出了该值，调度最小粒度和调度延迟两个目标则不可能被同时实现。

内核并没有提供参数来指定sched_nr_latency，它的值完全是由调度延迟和调度最小粒度来决定的。计算公式如下：

$$\text{sched_nr_latency} = \frac{\text{sysctl_sched_latency}}{\text{sysctl_sched_min_granularity}}$$

因此调度延迟是一个尽力而为的目标。当可运行的进程个数小于sched_nr_latency的时候，调度周期总是等于调度延迟（sysctl_sched_latency）。但是如果可运行的进程个数超过了sched_nr_latency，系统就会放弃调度延迟的承诺，转而保证调度最小粒度。在这种情况下调度周期等于最小粒度乘以可运行进程的个数，代码如下所示：

```
static u64 __sched_period(unsigned long nr_running)
{
    u64 period = sysctl_sched_latency;
    unsigned long nr_latency = sched_nr_latency; /* 进程个数过多，无法保证调度延迟，只能保证调度最小粒度 */

    /*
     * if (unlikely(nr_running > nr_latency)) {
     *     period = sysctl_sched_min_granularity;
     *     period *= nr_running;
     * }
     */
    return period;
}
```

上述函数并不难理解：

- 若运行队列中进程个数小于或等于sched_nr_latency，那么调度周期等于调度延迟。
- 若运行队列中进程个数大于sched_nr_latency，那么调度周期则等于可运行进程个数与调度最小粒度的乘积。

有了调度周期，我们就可以计算，分配给进程的运行时间了：

分配给进程的运行时间=调度周期*1/运行队列上进程个数

到目前为止，所有的讨论都是基于运行队列上所有的进程都有相同的优先级这个假设。但真实情况并非如此，有些任务优先级比较高，理应获得更多的运行时间。考虑到这种情况，完全公平调度又引入了优先级的概念。

完全公平调度通过引入调度权重来实现优先级，进程之间按照权重的比例，分配CPU时间。引入权重后，调度周期内分配给进程的运行时间的计算公式如下：

分配给进程的运行时间=调度周期*进程权重/运行队列所有进程权重之和

Linux下每一个进程都有一个nice值，该值的取值范围是[-20, 19]，其中nice值越高，表示优先级越低。默认的优先级是0。



注意 nice的英文含义是友好，nice值越高，表示越友好，越谦让，即优先级越低。具体说就是同等情况下，占有的CPU资源越少。

内核定义了一个数组，来表述每个不同nice值对应的权重：

```
static const int prio_to_weight[40] = {
/* -20 */ 88761, 71755, 56483, 46273, 36291,
/* -15 */ 29154, 23254, 18705, 14949, 11916,
/* -10 */ 9548, 7620, 6100, 4904, 3906,
/* -5 */ 3121, 2501, 1991, 1586, 1277,
/* 0 */ 1024, 820, 655, 526, 423,
/* 5 */ 335, 272, 215, 172, 137,
/* 10 */ 110, 87, 70, 56, 45,
/* 15 */ 36, 29, 23, 18, 15,
};
```

这个数组基本是通过如下公式来获得的：

```
weight = 1024 / (1.25 ^ nice_value)
```

其中普通进程的nice值等于0，其权重为基准的1024。nice值为0的进程权重被称为NICE_0_LOAD。当nice值为1时，权重等于1024/1.25，约等于820，当nice值为2时，权重等于1024/(1.25^2)。



注意 很有意思的是计算公式中的1.25是怎么来的？一般的概念是这样的，进程每降低一个nice值，将多获得10%的CPU时间。如果运行队列里有两个进程，一个nice值为0，另一个nice值为-1。那么按照约定，nice值为0的应该获得45%的CPU时间，而nice值为-1的应该获得55%的CPU时间。那么两者的权重比例应该是多少？

$$1/(1+x)=0.45$$

根据上面的计算公式，很容易算出，该值约等于1.222左右。内核计算时，选择该值为1.25。具体可阅读prio_to_weight定义出的注释。

Linux提供了如下函数来获取和修改进程的nice值：

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

两个系统调用的头两个参数都是which和who，这两个参数用于标识需要读取和修改优先级的进程。who参数如何解释，取决于which参数的值，具体如下：

- PRIO_PROCESS：操作进程ID为who的进程，如果who为0，那么使用调用者的进程ID。
- PRIO_PGRP：操作进程组ID为who的进程组的所有成员。如果who等于0，那么使用调用者的进程组ID。
- PRIO_USER：操作所有真实用户ID为who的进程。如果who等于0，使用调用者的真实用户ID。

getpriority函数返回which和who指定进程的nice值。如果存在多个进程符合指定的标准，那么返回优先级最高的那个nice值（即nice值最小的那个）。

因为进程优先级的范围为[-20, 19]，所以成功的时候，返回值也可能是-1。因此，不能用返回值是不是-1来判断调用是成功还是失败。正确的方法是，调用前将errno设置成0，然后调用getpriority函数。如果返回值是-1，并且errno不是0，才能确定调用失败。否则，调用成功。

```
errno = 0;

prio = getpriority(which,who);
if(prio == -1 && errno != 0)
{
    /*error handle*/
}
```

setpriority函数的返回值并不存在getpriority函数的困境。其成功时返回0，失败时返回-1，并置errno。常见的errno见表5-3。

表5-3 setpriority函数的出错情况及说明

errno	说 明
EACCESS	尝试获取更高的优先级（更低的 prio 值），但是没有 CAP_SYS_NICE 权限
EINVAL	which 的值不是 PRIO_PROCESS、PRIO_PGRP 或 PRIO_USER
ESRCH	which 和 who 指定的进程不存在
EPERM	指定进程的有效用户 ID 和调用进程的有效用户 ID 不一致，且调用进程没有 CAP_SYS_NICE 权限

对于其中的EACCESS错误码，这里仔细说明一下。在早期版本的Linux中非特权进程不能提升优先级，只能降低优先级。但在现在的Linux中，非特权进程也能适当地提升进程的优先级了。Linux提供了RLIMIT_NICE资源限制。如果一个进程的RLIMIT_NICE限制为25，那么其nice值可以提升到20-25=-5。详情可以查看getrlimit函数的手册。

调整进程的优先级会有什么影响？完全公平调度算法里，优先级比较高（nice值比较低）的进程会获得更多的CPU时间。

比如，有两个进程位于CPU的运行队列上，一个nice值是0（权重是1024），另外一个nice值是5（权重是335），按照前面的权重可以推算出，nice值为0的进程获得CPU的时间应该是nice值为5的3倍。

可以通过一个简单的测试来验证这个结论：

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <sched.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sched.h>
int heavy_work()
{
    double sum = 0.0;
    unsigned long long i = 0;
    while(1)
    {
        sum = sum + sin(i++);
    }
    return 0;
}
int main(int argc,char* argv[])
{
    cpu_set_t set ;
    CPU_ZERO(&set);
    CPU_SET(0,&set);
    int ret = sched_setaffinity(0,sizeof(cpu_set_t),&set);
    if(ret != 0 )
    {
        fprintf(stderr,"failed to bind the process to cpu 0 (%s)\n",
            strerror(errno));
        exit(1);
    }
    ret = fork();
    if(ret == 0)
    {
        errno = 0;
        ret = setpriority(PRIO_PROCESS,0,5);
        if(ret == -1 && errno != 0)
        {
            fprintf(stderr,"[%d] failed to change nice value (%s)\n",
                getpid(),strerror(errno));
            exit(1);
        }
    }
    heavy_work();
    return 0;
}
```

上面的程序设置了进程的CPU亲和力，父子进程都将运行在CPU 0上，不过，子进程首先调用setpriority函数将自己的nice值设置成了5，而父进程的nice值是默认值0。父子进程都是CPU bound型的程序，始终处于可运行状态。

```
manu@manu-rush:~$ ps -C nice_test -o pid,ppid,cmd,etime,nice,pri,psr
  PID   PPID  CMD               ELAPSED  NI  PRI  PSR
  3885    2695  ./nice_test        35:02    0   19    0
  3886    3885  ./nice_test        35:02    5   14    0
```

通过NI这一列可以看出，父进程的nice值是0，而子进程的nice值是5。父进程占用的CPU时间应该是子进程的三倍左右。通过/proc/PID/sched可以查看这些调度的信息，其中se_sum_exec_runtime的含义是累计运行的物理时间。

```
父进程

se.sum_exec_runtime      :      1584276.837760子进程

se.sum_exec_runtime      :      518296.243156
```

那么我们比较一下：

$$1024 \div 335 \approx 3.0567$$
$$1584276.837760 \div 518296.243156 \approx 3.0567$$

从执行时间上可以看出，执行时间几乎完美地符合权重比。原因就是决定每个进程运行时间片的时候，是根据权重来计算的。

有意思的是，如果CPU运行队列上的两个进程的nice值分别是10和15，那么两者占用的CPU时间的比例依然约等于3：1。原因是绝对的nice值并不影响调度决策，而是运行队列上进程间的优先级相对值，影响了CPU时间的分配。

5.4 完全公平调度的实现

上一节的全部内容，归纳起来就是下面这个公式：

分配给进程的运行时间 = 调度周期 * 进程权重 / 所有进程权重之和

但是上一节并没有介绍完全公平调度的算法实现，本节将尝试介绍完全公平调度的内容。完全公平调度的算法思想比较简单，按照CFS作者Ingo Molnar的总结：CFS百分之八十的工作可以用一句话来概括，那就是CFS在真实的硬件上模拟了完全理想的多任务处理器。

5.4.1 时间片和虚拟运行时间

在Linux操作系统中，每个CPU都维护有运行队列。在该队列上可能存在多个进程处于可执行状态，那么哪个进程应该先获得调度呢？

这个问题和生活中的某些问题很像。比如一个游戏机，5个小孩玩，当一个小孩玩完自己的时间片后，该由哪个小孩接着来玩呢？肯定有一个小孩跳出来：我玩的时间最短，应该由我来玩。这是非常朴素的思想，为每个小朋友玩的时间记账，玩的时间最短的小朋友将获得下一个玩的机会。

在进程优先级都相等的情况下，时间记账是一个非常好的方法，但是优先级的存在，给时间记账带来了一定的麻烦。有些进程优先级比较高，理应获得更多的CPU时间，这种情况下如何进行时间记账？

Linux引入了虚拟运行时间来解决这个记账的问题。假设CPU运行队列上有两个进程需要调度，`nice`值分别为0和5，两者的权重比是3: 1，调度周期为20毫秒。那么按照公式，第一个进程应该运行15毫秒，接着第二个进程运行5毫秒。尽管两个进程在调度周期内的实际运行时间不同，但是我们希望第一个进程的15毫秒和第二个进程的5毫秒，时间记账是相等的。即：

第一个进程15毫秒的记账值 = 第二个进程的5毫秒的记账值

这样两个进程就能根据时间记账值的大小交替执行了。这种时间加权记账的思想就是完全公平调度的核心了。

Linux内核定义了调度实体结构体，代码如下：

```
struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
    unsigned int on_rq;
    u64 exec_start;
    u64 sum_exec_runtime;
    u64 vruntime;
    u64 prev_sum_exec_runtime;
    u64 nr_migrations;
    ...
}
```

上述结构中，`sum_exec_runtime`维护的是真实时间记账信息。而`vruntime`维护的则是加权过的时间记账，即虚拟运行时间。

如何根据真实的时间计算出虚拟的运行时间，作为加权过的时间记账？公式如下。

$$\text{加权运行时间} = \text{真实运行时间} \times \frac{\text{NICE_0_LOAD}}{\text{进程权重}}$$

在该公式中，`NICE_0_LOAD`的值是`nice`值为0的进程的权重，即1024。前面的例子中，`nice`值为0的进程运行了15毫秒，因为其权重为1024，故其虚拟运行时间也为15毫秒；`nice`值为5的进程运行时间为5毫秒，因为其权重为335，所以记账时其虚拟运行时间为：

$$5 \times \frac{1024}{335} \approx 15$$

内核的`sched_slice`函数负责计算进程在本轮调度周期应分得的真实运行时间，其实现代码如下：

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    /* 本轮调度周期的时间长度 */

    /*
     * u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);
     * Linux支持组调度，所以此处有一个循环，
     */
}
```

*如果不考虑组调度，将调度实体简化成进程，会更好理解

```

*/
for_each_sched_entity(se) {
    struct load_weight *load;
    struct load_weight lw;
    cfs_rq = cfs_rq_of(se);
    load = &cfs_rq->load;
    if (unlikely(!se->on_rq)) {
        lw = cfs_rq->load;
        update_load_add(&lw, se->load.weight);
        load = &lw;
    }
    /*根据调度实体所占的权重，分配时间片的大小

*/
    slice = calc_delta_mine(slice, se->load.weight, load);
}
return slice;
}

```

在这个函数中，`calc_delta_mine`函数就是用来计算分配这个调度实体的时间片长度：

分配给进程的运行时间

=调度周期

* 进程权重

/ 所有进程权重之和

```
slice = calc_delta_mine(slice, se->load.weight, load);
```

在下一节中可以看到，内核会周期性地检查进程是不是已经耗完了自己的时间片，检查的方法就是判断进程本轮运行时间是否已经超过了`sched_slice`计算出来的时间片。如果超过，则表示运行时间足够久了，应该发生一次抢占。

更新进程虚拟运行时间的逻辑位于内核的`__update_curr`函数，该函数里更新了当前进程的真实运行时间和虚拟运行时间，同时也更新了CFS运行队列的最小虚拟运行时间。

```

static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{
    unsigned long delta_exec_weighted;
    schedstat_set(curr->statistics.exec_max,
                  _max((u64)delta_exec, curr->statistics.exec_max));
    /*更新进程的真实运行时间

*/
    curr->sum_exec_runtime += delta_exec;
    schedstat_add(&cfs_rq, exec_clock, delta_exec);
    /*calc_delta_fair用来计算加权后的运行时间

*/
    delta_exec_weighted = calc_delta_fair(delta_exec, curr);
    /*更新进程的虚拟运行时间

*/
    curr->vruntime += delta_exec_weighted;
    /*更新运行队列的最小虚拟运行时间

*/update_min_vruntime(cfs_rq);
#ifdef CONFIG_SMP && defined CONFIG_FAIR_GROUP_SCHED
    cfs_rq->load_unacc_exec_time += delta_exec;
#endif
}

```

运行队列的最小虚拟运行时间是什么？为什么需要它？

运行队列上存在多个进程，随着时间的流逝，每个进程的虚拟时间各不相同，内核会将所有进程中虚拟运行时间的最小值记录到运行队列的最小虚拟运行时间（`vruntime`）中。当然运行队列的最小虚拟运行时间是奔流向前的，只会单调增大，绝不会减小。

为什么要维护这个值？CFS算法可确保队列上的所有进程步调一致地轮流运行，虚拟运行时间不断增大，大部分进程的虚拟运行时间相差也不会太远。但是记录下队列虚拟运行时间的最小值仍然是有意义的。比如新加入一个进程，应该给它的虚拟运行时间赋初始值，初始值应是多少？再比如进程陷入了漫长的休眠，醒来时已经沧海桑田，相对其他进程，它的虚拟运行时间已经大幅落后。内核应该将该进程的虚拟运行时间调整成何值？又比如内核不得不将某个进程从一个CPU的运行队列拉到另一个CPU的运行队列中，该进程的虚拟运行时间该如何调整？此时，维护运行队列的最小虚拟运行时间的意义就彰显出来了。运行队列的最小虚拟运行时间给了我们一个基准，根据这个基准值可以知道，该CPU运行队列上的大部分进程的虚拟运行时间就在该值附近，且大于该值。在后面分析新创建的进程和唤醒休眠进程时，会分析内核如何调整这些进程的虚拟运行时间。

进程有了虚拟运行时间，完全公平调度器挑选下一个运行程序时就变得非常简单了，只需要挑选具有最小虚拟运行时间（`vruntime`）的进程投入运行即可。这就是完全公平调度算法的核心所在。

内核为了加速挑选具有最小虚拟运行时间的进程，使用了红黑树数据结构。运行队列上的所有调度实体都是红黑树的节点。红黑树是平衡二叉树的一种，调度实体的虚拟运行时间是红黑树的键值。虚拟运行时间最小的调度实体，位于红黑树的最左端。因此挑选下一个运行程序，就简化成了从红黑树上取出最左端的节点（如图5-11所示）。

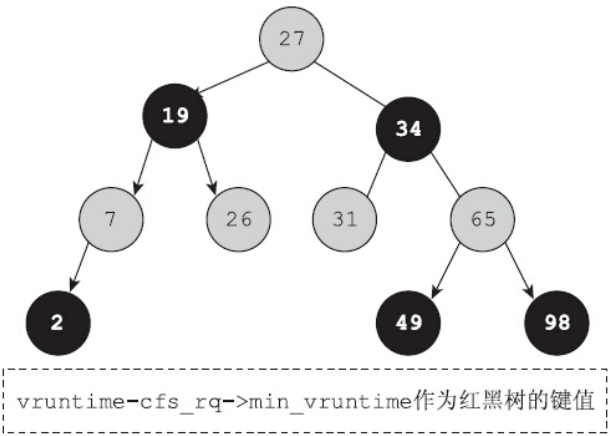


图5-11 调度器根据虚拟运行时间（`vruntime`）将进程在红黑树中排序

维护进程的虚拟运行时间就成了调度算法的关键。问题是何时会更新进程的虚拟运行时间呢？可以查看内核代码中所有调用 `update_curr` 的函数。内核会周期性地更新进程的虚拟运行时间，也会在某些合适的时间点调用 `update_curr` 更新。我们暂时强忍好奇，继续探索。在探索的过程中，会多次遇到调用 `update_curr` 的函数。

5.4.2 周期性调度任务

周期性调度任务是调度框架中很重要的一个部分。因为Linux是抢占式多任务，系统需要周期性地检查，当前运行的进程是不是已经耗尽了它的时间片，是不是应该发起一次抢占了。这就是周期性调度任务的职责。

当时钟发生中断时，首先调用的是tick_handle_peroid函数。该函数会调用scheduler_tick函数，而scheduler_tick函数是进程调度框架中的重要函数，负责处理进程调度相关的周期性任务，如图5-12所示。

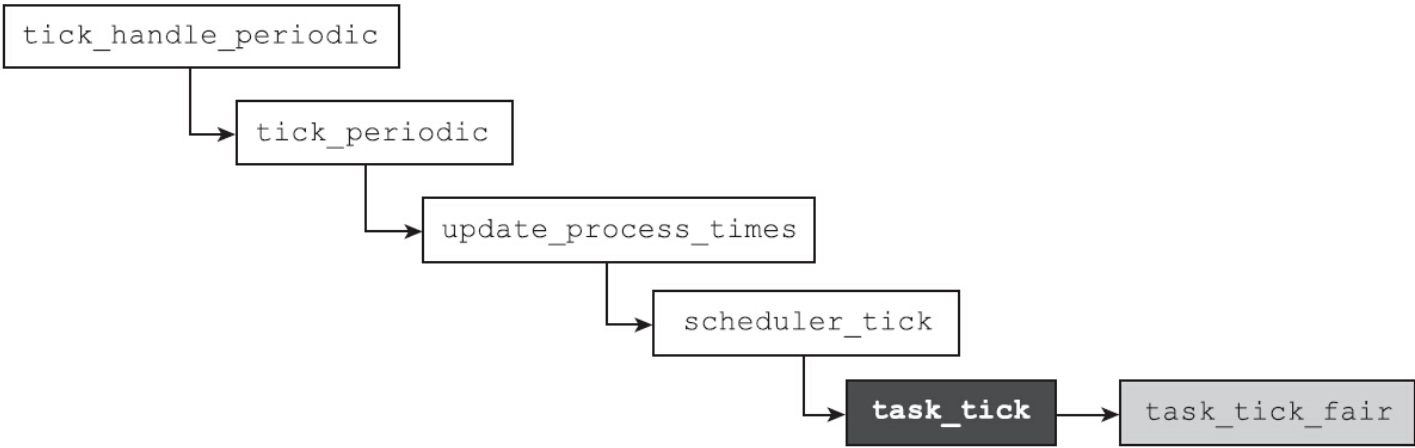


图5-12 scheduler_tick函数的调用栈关系

在scheduler_tick函数中一个非常重要的调用是：

```
curr->sched_class->task_tick(rq, curr, 0);
```

在Linux的实现中调度器采用了模块化的实现，任何一个调度类，都要实现task_tick这个函数。那这个task_tick函数要完成哪些使命呢？主要的工作是更新当前运行进程调度相关的统计信息，以及判断是否需要发生调度。

对于完全公平的调度而言，task_tick函数为：

```
.task_tick      = task_tick_fair,
```

在task_tick_fair函数中，内核更新了正在运行的进程的时间统计，包括真实运行时间和虚拟运行时间，代码如下：

```
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;
    /*为了支持组调度，引入了调度实体的概念*/

    /*
     * for_each_sched_entity(se) {
     *     cfs_rq = cfs_rq_of(se);
     *     entity_tick(cfs_rq, se, queued);
     * }
     */
    static void
    entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
    {
        /*更新正在运行进程的统计信息*/

        /*
         * update_curr(cfs_rq);
         * update_entity_shares_tick(cfs_rq);
         * ...
         * /*如果可运行状态的进程个数大于
         *
         * 1. 检查是否可以抢占当前进程
         *
         * if (cfs_rq->nr_running > 1)
         *     check_preempt_tick(cfs_rq, curr);
         */
    }
}
```

在我们探索的第一站就遇到了更新`updat_curr`的地方。时钟中断触发了周期性的调度任务，其中一项重要的任务就是通过`updat_curr`函数更新调度的统计信息。它随着时钟中断处理函数周期性地执行，更新进程的虚拟运行时间、真实运行时间和运行队列的最小虚拟运行时间等。

内核需要知道在什么时候调用`schedule`函数，而不能仅仅依靠用户程序显式地调用`schedule`函数。如果将`schedule`函数的发起完全委托给用户程序，那么用户程序可能会无止尽地执行下去，而导致其他进程饿死。内核提供了一个`need_resched`标志位来表明是否需要重新执行一次调度。很明显，伴随着时钟中断发生的周期性调度任务是一个非常好的时机来判断当前进程是否应该被抢占（另一个时机是进程从睡眠状态醒来时，`try_to_wake_up`函数也会判断是否需要设置`need_resched`标志位来抢占当前的进程）。

当运行队列上处于可运行状态的进程不止一个时，内核会调用`check_preempt_tick`函数来检查是否应该发生抢占。该函数确保了当前进程使用完自己的时间片后，可以及时地让出CPU，代码如下：

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;
    struct sched_entity *se;
    s64 delta;
    /*ideal_runtime记录进程应该运行的时间

*/
    ideal_runtime = sched_slice(cfs_rq, curr);
    /* delta_exec记录进程真实运行的时间

*/
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    /*如果实际运行时间超过了应该运行的时间，则需要调度出去，被抢占

*/
    if (delta_exec > ideal_runtime) {
        /*resched_task 会负责设置

need_resched标志位

*/
        resched_task(rq_of(cfs_rq)->curr);
        clear_buddies(cfs_rq, curr);
        return;
    }
    ...
    /* 如果当前进程运行时间低于调度的最小粒度，则不允许发生抢占

*/
    if (delta_exec < sysctl_sched_min_granularity)
        return;
    ...
}
```

在`check_preempt_tick`中可以看出，进程有自己的完美运行时间，即本轮调度周期应得的时间片。如果本轮执行时间已经超出了时间片，就会执行`resched_task`函数，在该函数中会通过`set_tsk_need_resched`函数来设置`need_resched`标志位，告诉内核请尽快调用`schedule`函数。如果进程的本轮运行时间小于调度最小粒度，那么不允许发生抢占。

`resched_task`函数仅仅是设置标志位，并没有真正地执行进程切换。进程调度发生的时机之一是发生在中断返回时，`check_preempt_tick`函数是`scheduler_tick`函数的一部分，而`scheduler_tick`函数是中断处理程序的一部分。执行完中断处理，会检查`need_resched`标志位是否置位，如果置位，那就自然会调用`schedule`函数来执行切换。

5.4.3 新进程的加入

刚创建的普通进程，它的虚拟运行时间是0吗？

这个问题的答案很明显，如果新创建进程的vruntime是0，那么它的值会比已经长时间运行的进程的虚拟运行时间小很多。它会在相当长的时间内保持着调度的优势，一直运行。这显然是不合理的。

为了系统地回答上面的问题，我们跟踪下新进程出生之后，发生了哪些事情，图5-13是在创建新进程的过程中与进程调度有关系的流程。

首先分析一下sched_fork的内核代码：

```
void sched_fork(struct task_struct *p)
{
    unsigned long flags;
    int cpu = get_cpu();
    /*初始化调度相关的值，如调度实体、运行时间、虚拟运行时间等

*/
    __sched_fork(p);
    /*设置成

TASK_RUNNING, 其实新创建的进程并没有真正地在

CPU上执行,

*此举的目的是防止外部信号和时间将其唤醒，之后插入运行队列

*/
    p->state = TASK_RUNNING;
    p->prio = current->normal_prio;
    /*如果设置了

sched_reset_on_fork标志位，后面会讨论

*/
    if (unlikely(p->sched_reset_on_fork)) {
        if (task_has_rt_policy(p)) {
            p->policy = SCHED_NORMAL;
            p->static_prio = NICE_TO_PRIO(0);
            p->rt_priority = 0;
        } else if (PRIO_TO_NICE(p->static_prio) < 0)
            p->static_prio = NICE_TO_PRIO(0);
        p->prio = p->normal_prio = __normal_prio(p);
        set_load_weight(p);
        p->sched_reset_on_fork = 0;
    }
    /*如果不是实时进程，则调度类为完全公平调度类

*/
    if (!rt_prio(p->prio))
        p->sched_class = &fair_sched_class;
    /*如果调度类实现了

task_fork函数，则调用该函数

*/
    if (p->sched_class->task_fork)
        p->sched_class->task_fork(p);
    ...
}
```

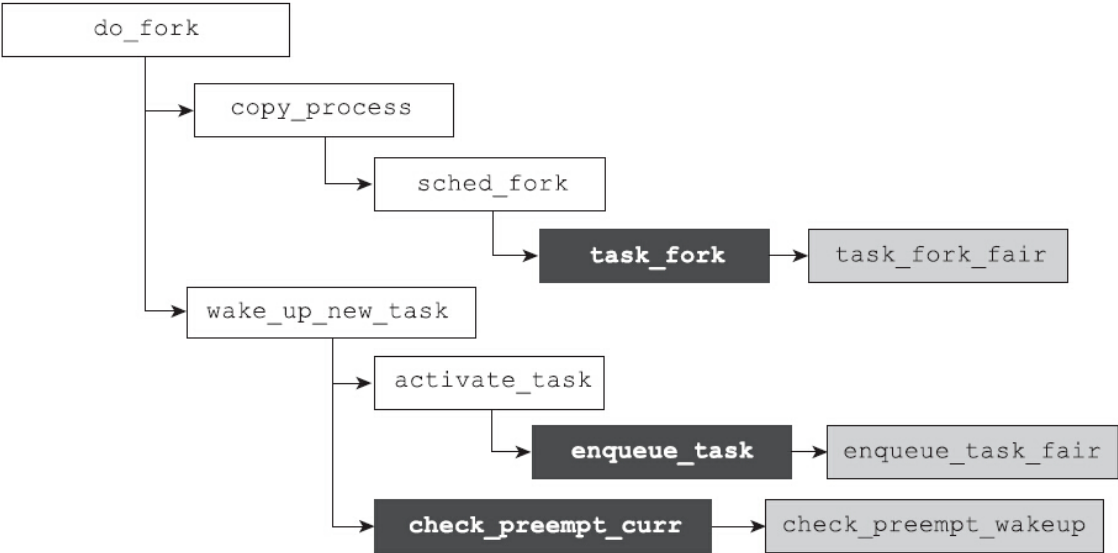


图5-13 创建新进程中与调度相关的函数

`sched_fork`函数的主要工作是初始化进程的与调度相关的变量，确定进程所属的调度类及优先级设置。根据进程所属的调度类，执行与调度类相关的函数。

调度类需要实现`task_fork`这个hook函数。该函数用于处理与新创建的进程相关的初始化事宜。对于完全公平调度类，该函数的实现为：

```
.task_fork    = task_fork_fair,
```

下面一起走进`task_fork_fair`函数来看一下完全公平调度类是如何处理新创建的进程的：

```
static void task_fork_fair(struct task_struct *p)
{
    struct cfs_rq *cfs_rq = task_cfs_rq(current);
    struct sched_entity *se = &p->se, *curr = cfs_rq->curr;
    int this_cpu = smp_processor_id();
    struct rq *rq = this_rq();
    unsigned long flags;
    raw_spin_lock_irqsave(&rq->lock, flags);
    update_rq_clock(rq);
    if (unlikely(task_cpu(p) != this_cpu)) {
        rcu_read_lock();
        set_task_cpu(p, this_cpu);
        rcu_read_unlock();
    }
    /*更新

CFS调度类的队列

, 包括执行

__update_curr更新当前进程统计

*/
    update_curr(cfs_rq);
    /*新创建进程的

vruntime初始化成父进程的

vruntime.

*紧随其后的

place_entity函数会负责调整新创建进程的
```

```
vruntime*/
    if (curr)
        se->vruntime = curr->vruntime;
    place_entity(cfs_rq, se, 1);
    /*如果设置了子进程先运行，并且父进程的

vruntime小子子进程，则交换彼此的

vruntime，确保子进程先执行

*/
    if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) {
        swap(curr->vruntime, se->vruntime);
        resched_task(rq->curr);
    }
    /*此处减去当前运行队列的最小虚拟运行时间，

*真正进入运行队列，即执行

enqueue_entity时，

*进程的

vruntime会加上

cfs_rq->vruntime*/
    se->vruntime -= cfs_rq->min_vruntime;
    raw_spin_unlock_irqrestore(&rq->lock, flags);
}
```

关于新进程，进程调度领域有两大悬疑：

- 新进程的虚拟运行时间到底是多少？
- 父子进程中哪个先执行？

下面将分别进行分析。

1.新创建进程的虚拟运行时间初始值

`task_fork_fair`函数中有以下内容：

```
if (curr)
    se->vruntime = curr->vruntime;
place_entity(cfs_rq, se, 1);
```

从上面的函数中可以看出，新创建子进程的虚拟运行时间首先被初始化成父进程的虚拟运行时间，接下来会调用了`place_entity`函数，而`place_entity`函数会调整新创建进程的虚拟运行时间。

“`place_entity`”，直白的翻译就是放置调度实体的意思，即把调度实体放置到合适的位置。如何才能决定调度实体的位置呢？毫无疑问，只能通过调整调度实体的虚拟运行时间来实现。

`place_entity`函数用来处理两种比较特殊的情况：

- 调整新创建进程的虚拟运行时间。
- 调整从休眠中唤醒进程的虚拟运行时间。

这两种情况根据该函数的第三个参数initial来区分。initial等于1则表示调整新创建进程的虚拟运行时间。

下面来看看place_entity函数是如何调整新创建进程的虚拟运行时间的，代码如下：

```
static void
place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
    u64 vruntime = cfs_rq->min_vruntime;
    if (initial && sched_feat(START_DEBIT))
        vruntime += sched_vslice(cfs_rq, se);

    vruntime = max_vruntime(se->vruntime, vruntime); se->vruntime = vruntime;
}
static u64 sched_vslice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    return calc_delta_fair(sched_slice(cfs_rq, se), se);
}
```

完全公平调度类的运行队列cfs_rq中维护有成员变量min_vruntime，该变量存放的是此运行队列中的最小虚拟运行时间。就像前面所说的，它提供了一个基准值，通过它我们无须遍历队列上所有进程的虚拟运行时间，就可以得知该运行队列的整体情况了。大多数进程的虚拟值在该值附近，且略大于该值。

内核提供了很多调度的特性，记录在/sys/kernel/debug/sched_features中，如下所示：

```
cat /sys/kernel/debug/sched_features
GENTLE_FAIR_SLEEPERS START_DEBIT

NO_NEXT_BUDDY LAST_BUDDY_CACHE HOT_BUDDY_WAKEUP
PREEMPTION_ARCH_POWER NO_HRTICK NO_DOUBLE_TICK LB_BIAS NONTASK_POWER TTWU
QUEUE_NO_FORCE_SD_OVERLAP RT_RUNTIME_SHARE NO_LB_MIN NUMA_NUMA_FAVOUR_HIGHER
NO_NUMA_RESIST_LOWER
```

其中START_DEBIT特性是用来给新创建的进程略加惩罚的。如果没有START_DEBIT选项，子进程的虚拟运行时间为：

```
max (父进程的虚拟运行时间,

CFS运行队列的最小运行时间

)
```

这个值通常比较小，这就意味着子进程很快就能获得调度的机会，因此也就给了恶意进程可乘之机。因为恶意进程可以通过不停地fork来获得更多的CPU时间。如果设置了START_DEBIT选项，会通过增大子进程的虚拟运行时间来惩罚新创建的进程，使新创建的进程晚一点才能获得被调度的机会。

那么虚拟运行时间增大多少呢？看看下面的语句：

```
vruntime += sched_vslice(cfs_rq, se);
```

前面介绍过sched_slice函数是用来计算进程的时间片的，那么sched_vslice函数又是何意呢？

```
static u64 sched_vslice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    return calc_delta_fair(sched_slice(cfs_rq, se), se);
}
```

sched_vslice函数是根据时间片的值，来计算对应的虚拟时间片的值。即根据进程的优先级来调整。调整的算法前面已经提到过了。

打开了START_DEBIT特性，子进程的虚拟运行时间就会被初始化成：

```
max (父进程的虚拟运行时间,
```

CFS运行队列的最小运行时间

+进程虚拟时间片

)

2.父子进程谁先执行

另一大悬念是父子进程哪个会先执行？

内核提供了配置选项`sched_child_runs_first`，该值记录在：

```
/proc/sys/kernel/sched_child_runs_first
```

该配置选项是1的话，`fork`之后子进程将优先获得调度，如果是0的话，父进程将优先获得调度。内核版本自2.6.32开始，该值默认是0，即父进程优先执行。

`task_fork_fair`函数中有以下代码：

```
if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) {
    swap(curr->vruntime, se->vruntime);
    resched_task(rq->curr);
}
```

如果要设置子进程优先获得调度，则会通过`entity_before`函数来比较父子进程的`vruntime`，如果父进程的`vruntime`小，则需要和子进程互换`vruntime`以确保子进程优先获得调度。

但是正如Linux在邮件 [\[1\]](#) 中提到的，无论是父进程先运行还是子进程先运行，内核控制选项提供的是一种倾向或偏好（`preference`），而不是一种保证（`guarantees`） [\[2\]](#)。在编写应用程序时，无论内核参数`sched_child_runs_first`为何值，都不能作为作为父进程或子进程先运行的保证，如果需要保证运行次序，程序需要使用其他同步方法来确保运行的次序。

分析完两大悬疑，继续分析`task_fork_fair`函数。在该函数中有一条语句非常奇怪，该语句代码如下：

```
se->vruntime -= cfs_rq->min_vruntime;
```

为何要减掉运行队列的最小虚拟运行时间？继续向下看就可以恍然大悟了。因为在`do_fork`的末尾会调用`wake_up_new_task`函数（如图5-14所示）。事实上在对称多处理器结构上，新创建的进程和父进程不一定在同一个CPU上运行。进程刚刚创建好，尚未运行，这是多个CPU之间负载均衡的一个良机。Linux也是这么做的，在`wake_up_new_task`函数中会首先调用如下语句，选择一个合适的CPU：

```
set_task_cpu(p, select_task_rq(p, SD_BALANCE_FORK, 0));
```

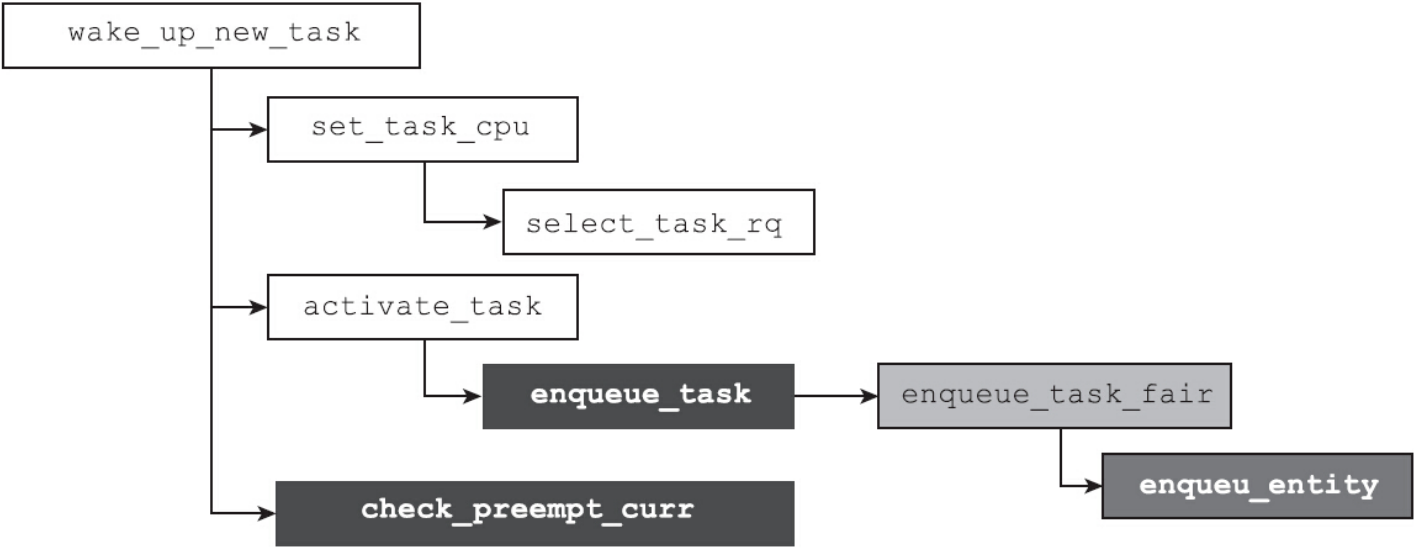


图5-14 wake_up_new_task函数

很不幸的是，不同的CPU之间负载并不完全相同，有的CPU更忙一些，而且每个CPU都有自己的运行队列cfs_rq，不同的CPU运行队列的最小虚拟运行时间min_vruntime并不相同。如果新创建的进程从一个CPU的运行队列迁移到另外一个CPU的运行队列，就可能会产生问题。比如新创建的进程从min_vruntime小的CPU A跳到min_vruntime非常大的CPU B，它就会占便宜，因为它的虚拟运行时间会在相当长的时间范围内都是最小的，从而产生调度的不公平。

解决的方法非常简单：



enqueue_task也是调度类的hook函数，每一个调度类都要实现该函数，对于完全公平的调度而言：

```
.enqueue_task = enqueue_task_fair,
```

在enqueue_task_fair函数中，会调用enqueue_entity函数，在该函数中有以下语句和task_fork_fair函数相呼应：

```
static void task_fork_fair(struct task_struct *p)
{se->vruntime -= cfs_rq->min_vruntime;
```



```
}
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    ...if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
        se->vruntime += cfs_rq->min_vruntime;...
}
```

事实上该解决方案不仅仅只是用于新创建的进程这一个场景。Linux支持CPU之间的负载均衡，可以将进程从一个CPU迁移到另外一个CPU，为了防止不公平的产生，也采用了上述的解决方案。

```
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    ...
    if (!(flags & DEQUEUE_SLEEP))
        se->vruntime -= cfs_rq->min_vruntime;
    ...
}
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    ...
    if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
        se->vruntime += cfs_rq->min_vruntime;...
}
```

创建新的进程不仅是CPU之间负载均衡的良机，也是检测是否可以发生抢占的良机。因此，`wake_up_new_task`在最后会调用`check_preempt_curr`函数。该函数会负责检查可否抢占当前的运行进程。这个函数的详细内容，将放到下一节来分析。

[1] Re:[GIT PULL]sched/core for v2.6.32:<http://thread.gmane.org/gmane.linux.kernel/888423/focus=888543>。

[2] <http://stackoverflow.com/questions/17391201/does-proc-sys-kernel-sched-child-runs-first-work>。

5.4.4 睡眠进程醒来

睡眠进程的虚拟运行时间会保持不变吗？

如何对待睡眠进程也是调度器需要解决的问题。因为交互型的进程会不断陷入休眠状态中，并等待用户的输入。虽然这类进程对CPU的整体消耗并不大，但是要求响应必须及时，否则用户会感觉到系统卡顿，用户体验就会很糟糕。

对CFS之前的O（1）调度器来说，交互型进程堪称其阿喀琉斯之踵。该调度算法的交互进程识别启发式算法异常复杂，该启发式算法融入了睡眠时间作为考量的标准，但是对于一些特殊的情况，经常判断不准，而且经常是改完一种情况又发现另一种特殊情况。

CFS调度算法并没有刻意地区分交互型进程和批处理型进程，依然漂亮地满足了交互型进程需要及时响应的需求。CFS算法是如何做到的呢，对于从休眠中醒来的进程，CFS进行了哪些处理呢？这是本节要介绍的内容。

睡眠进程和等待队列的关系在5.1节已经介绍过。当内核调用wake_up系列宏时，会执行运行队列元素中指定的回调函数，而回调函数通常是default_wake_function。该函数是try_to_wake_up的简单封装，因此当进程被内核唤醒时，内核通常会执行try_to_wake_up函数。

概括地讲，try_to_wake_up函数的职责是：

- 1) 把从休眠中醒来的进程放到合适的运行队列。
- 2) 将进程的状态设置为TASK_RUNNING。
- 3) 判断醒来的进程是否应该抢占当前正在运行的进程，如果是，则设置need_resched标志位。

try_to_wake_up的部分流程如图5-15所示。

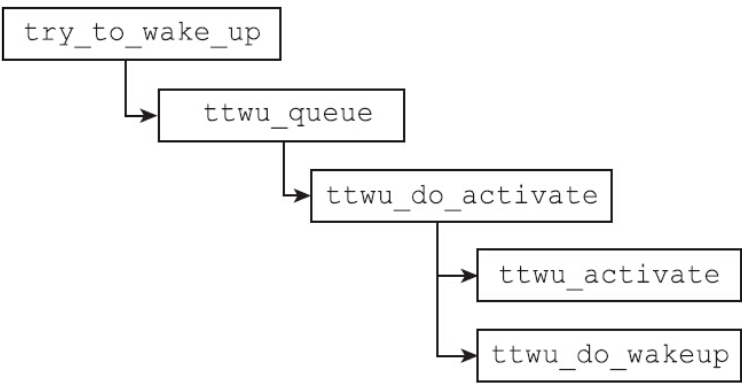


图5-15 try_to_wake_up函数

前面提到的try_to_wake_up负责的三件事，分别由以下函数负责完成，如表5-4所示。

表5-4 try_to_wake_up三个主要任务及对应的负责函数

事 件	相 关 函 数
将醒来的进程放入合适的运行队列中	ttwu_activate
设置进程状态为 TASK_RUNNING	ttwu_do_wakeup
判断唤醒进程能否抢占当前的进程，是则置 need_resched	ttwu_do_wakeup

首先来看看ttwu_active函数的相关内容：

```
static void ttwu_activate(struct rq *rq, struct task_struct *p, int en_flags)
{
    activate_task(rq, p, en_flags);
    p->on_rq = 1;
    /* if a worker is waking up, notify workqueue */
    if (p->flags & PF_WQ_WORKER)
        wq_worker_waking_up(p, cpu_of(rq));
}
static void activate_task(struct rq *rq, struct task_struct *p, int flags)
{
}
```

```
if (task_contributes_to_load(p))
    rq->nr_uninterruptible--;
/*将进程插入运行队列。

enqueue_task是调度类

hook函数

*/
enqueue_task(rq, p, flags);
}
static void enqueue_task(struct rq *rq, struct task_struct *p, int flags)
{
    update_rq_clock(rq);
    sched_info_queued(p);
    /*根据进程所属的调度类，执行相应的

enqueue_task函数

*/
p->sched_class->enqueue_task(rq, p, flags);
}
```

其执行脉络如图5-16所示。

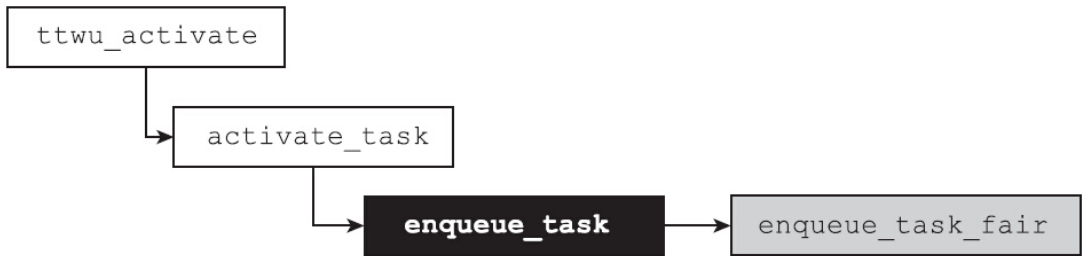


图5-16 ttwu_activate函数

activate_task函数和deactivate_task函数一样，都是调度框架内的重要函数，并且两者是一对，就好像wake_up和wait_event是一对一样。当进程调用wait_event时，进程从可运行状态变成睡眠状态，因此需要通过deactivate_task函数将进程从运行队列中移除，与此对应的，当内核调用wake_up函数把进程从休眠状态唤醒时，内核需要通过activate_task函数将进程放入运行队列中。如果对5.4.3节创建新进程还有印象的话，可以看到无论是创建新进程，还是唤醒休眠进程，都会执行到该函数，如图5-17所示。

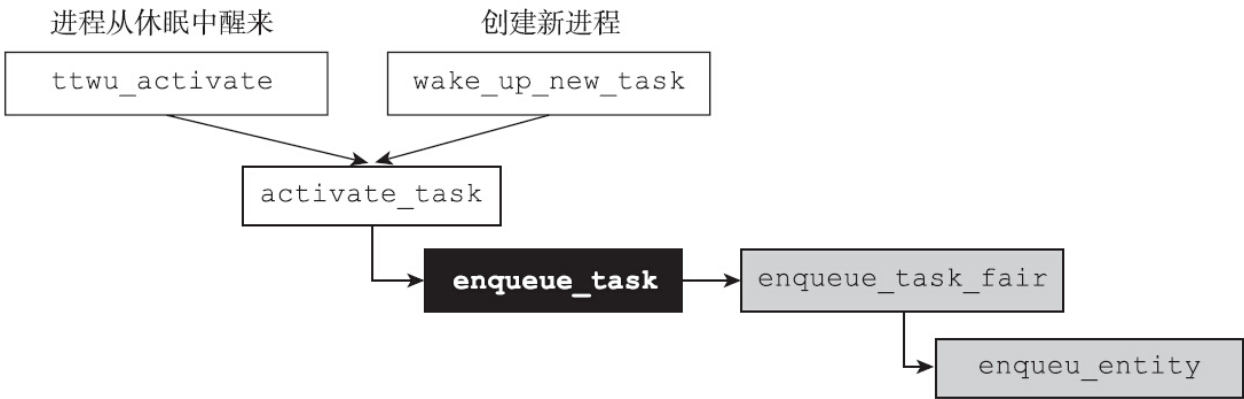


图5-17 activate函数相关流程

其中enqueue_task函数是调度类的hook函数，每个调度类都需要实现该函数。其含义顾名思义，即将进程放入运行队列。对于完全公平调度类而言，该函数指针指向的是enqueue_task_fair，代码如下：

```
.enqueue_task = enqueue_task_fair,
```

enqueue_task_fair很大部分的工作是更新调度相关的统计，其中有一支代码路径非常有意思（如图5-18所示），下面将重点介绍。

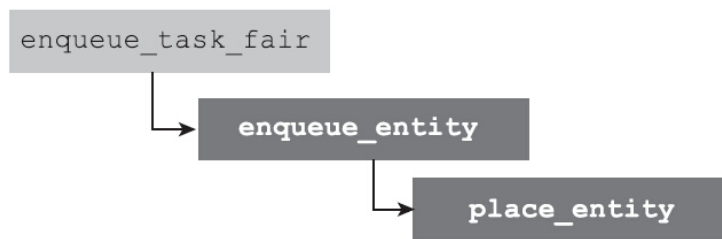


图5-18 CFS的enqueue_task_fair函数

这条路径之所以很重要，是因为它决定了休眠进程醒来后的虚拟运行时间。回到本节开头的问题：休眠进程的虚拟运行时间会保持不变吗？答案是否定的。很多进程可能会长时间地休眠，在这个过程中，如果虚拟运行时间vruntime保持不变，一旦该进程醒来，它的vruntime就会比运行队列上的其他进程小很多，因为会长时间保持调度的优势。这显然是不合理的。对于这种情况，完全公平调度的做法是，以运行队列的min_vruntime为基础，给予一定的补偿。

补偿多少？这就又要去看看我们的老朋友place_entity函数了。在创建新进程时，曾经走到过该函数，那时该函数负责决定新进程的虚拟运行时间。下面来看看对于被唤醒的休眠进程，该函数是如何决定进程的虚拟运行时间的：

```

static void
place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
    u64 vruntime = cfs_rq->min_vruntime;
    ...
    /*从休眠中醒来

*/
    if (!initial) {
        /*补偿一个调度周期

*/
        unsigned long thresh = sysctl_sched_latency;
        /*如果设置了

GENTLE_FAIR_SLEEPERS,则补偿半个调度周期

*/
        if (sched_feat(GENTLE_FAIR_SLEEPERS))
            thresh >>= 1;
        vruntime -= thresh;
    }
    vruntime = max_vruntime(se->vruntime, vruntime);
    se->vruntime = vruntime;
}
  
```

当initial等于0时，表示正在处理从休眠中醒来的进程。如果没有设置GENTLE_FAIR_SLEEPERS特性，那么在队列最小虚拟运行时间的基础上，补偿1个调度延迟，如果设置了GENTLE_FAIR_SLEEPERS，那么补偿减半，即补偿半个调度延迟。默认情况下，GENTLE_FAIR_SLEEPER的特性是打开的。

但休眠进程醒来后的虚拟运行时间并非只是简单粗暴地设置成队列的最小运行时间减掉补偿值。影响因素还有进程原本的虚拟运行时间，如下所示：

```

vruntime = max_vruntime(se->vruntime, vruntime);
  
```

如果休眠进程的睡眠时间非常短，很有可能进程原本的虚拟运行时间要大于上述计算得到的值，此时，休眠进程的虚拟运行时间不变，即为睡眠前的值。如果休眠进程的睡眠时间特别久，醒来时已经沧海桑田，那么就将虚拟运行时间设置为所在运行队列的最小虚拟运行时间减去补偿量。

从上面的代码可以看出，从长时间休眠中醒来的进程，因为其虚拟运行时间较小（比队列的最小虚拟运行时间还小），所以会获得优先的调度，从而使交互型进程得到及时的响应。



说明 这种对休眠进程进行奖励的做法，在进程调度设计领域存在一定的争议。内核进程调度领域的大牛Con Kolivas就坚持认为，调度器只需要向前看，而不应该考虑一个进程的过去。在早期的CFS调度算法（版本2.6.23）中，CFS会负责记录进程的sleep time，2.6.24版本之后的内核，就不再考虑进程过去的睡眠时间了。

但是CFS做得并不彻底，在`place_entity`函数中，对休眠进程进行了补偿。在CFS早期的版本中，`sleeper fairness`的特性会导致在一些情况下出现严重的调度延迟。在Jens Axboe的测试中^[1]，甚至会出现10秒的延迟，也有客户报告在编译内核时，音频视频会有严重的停顿。上面代码中的`GENTLE_FAIR_SLEEPER`特性就是作者Ingo给出的Patch，这个特性解决了10秒的延迟和其他鼠标滞后、视频停顿等交互性的问题。

[1] Re: BFS vs.mainline scheduler benchmarks and measures:<http://lwn.net/Articles/352875/>。

5.4.5 唤醒抢占

无论是try_to_wake_up唤醒睡眠的进程还是wake_up_new_task唤醒新创建的进程，内核都会使用check_preempt_curr函数来检查新唤醒的进程或新创建进程是否可以抢占当前运行的进程，如图5-19所示。

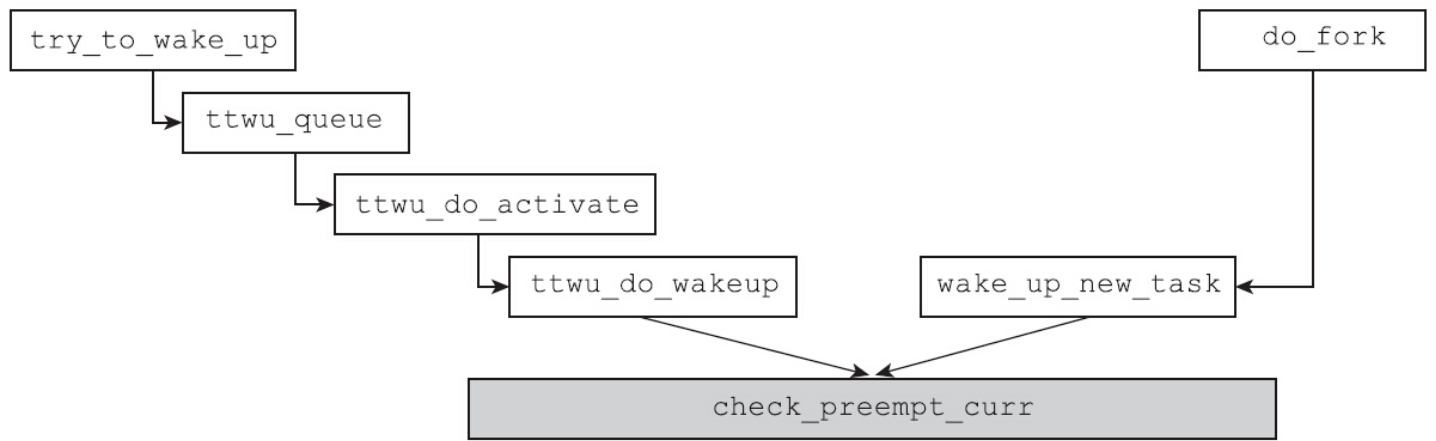


图5-19 唤醒抢占

判断是否应该抢占的工作被委托给了check_preempt_curr函数来完成。

```
static void check_preempt_curr(struct rq *rq, struct task_struct *p, int flags)
{
    const struct sched_class *class;
    if (p->sched_class == rq->curr->sched_class) {
        rq->curr->sched_class->check_preempt_curr(rq, p, flags);
    } else { /*for_each_class. 从高优先级的调度类到低优先级的调度类

    /*
     * for each class(class) {
     *     /*如果候选进程的调度类低于当前进程所属的调度类，就直接跳出。
     *
     * 不许低优先级的调度类抢占高优先级的调度类
     *
     *     if (class == rq->curr->sched_class)
     *         break;
     *     /*如果候选进程所属的调度类优先级高于当前进程的调度类，
     *
     * 则通过执行
     *
     * resched_task函数，设置
     *
     * need_resched标志位
     *
     *     if (class == p->sched_class) {
     *         resched_task(rq->curr);
     *         break;
     *     }
     * }
     * if (rq->curr->on_rq && test_tsk_need_resched(rq->curr))
     *     rq->skip_clock_update = 1;
     * }
```

判断能否发生抢占的逻辑异常简单，也是符合正常人思维的：

- 如果候选进程和正在运行的进程属于同一个调度类，那么调度类内部提供方法解决。
- 如果候选进程和正在运行的进程属于不同的调度类，候选进程所属调度类的优先级高于正在运行进程的调度类的优先级，则可以抢占，否则不可以。

注意新唤醒的进程不一定是普通进程，也可能是实时进程。如果唤醒的进程是实时进程而当前运行的进程为普通进程，则会设置 `need_resched` 标志位，因为实时进程总是会抢占CFS调度域的普通进程。

每一种调度类都应该实现自己的 `check_preempt_curr` 函数来判断是否需要发生抢占，对于完全公平调度类，`check_preempt_curr` 的实现为 `check_preempt_wakeup` 函数。

```
.check_preempt_curr = check_preempt_wakeup,
```

如果候选进程和正在运行的进程都属于完全公平调度类，那么候选进程到底会不会抢占当前运行的进程呢？哪些因素会影响到抢占行为呢？

如果被唤醒的进程的睡眠时间非常久（上百毫秒、几百毫秒、几秒甚至更久），前面的 `place_entity` 函数会将睡眠进程的虚拟运行时间设置为队列的最小虚拟运行时间减掉补偿的半个调度周期，这会使睡眠进程的虚拟运行时间非常的小，醒来时几乎总是会抢占当前的进程，这种行为也是期待的行为，因为它可以保证交互型进程的响应时间。

但是也有很多进程的睡眠时间非常短暂（比如只有几毫秒甚至更短），醒来之后通过 `place_entity` 函数计算得出的虚拟运行时间值仍然是自己本来的虚拟运行时间值。如果仅仅比较醒来的进程和当前运行进程的虚拟运行时间来决定是否抢占，那么很可能会使得抢占过于频繁^[1]。因此Linux引入了唤醒抢占粒度 `sched_wakeup_granularity_ns`，可以通过如下方法来查看系统的唤醒抢占粒度 `sched_wakeup_granularity_ns` 的值：

```
cat /proc/sys/kernel/sched_wakeup_granularity_ns
2000000
```

引入该最小粒度后，唤醒进程抢占当前进程的条件是：只有当唤醒进程的 `vruntime` 小，并且两者的差值 `vdiff` 大于 `sched_wakeup_granularity_ns` 时，才能抢占。具体的算法实现如下：

```
static int
wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
{
    s64 gran, vdiff = curr->vruntime - se->vruntime;
    if (vdiff <= 0)
        return -1;
    gran = wakeup_gran(curr, se);
    if (vdiff > gran)
        return 1;
    return 0;
}

static unsigned long
wakeup_gran(struct sched_entity *curr, struct sched_entity *se)
{
    unsigned long gran = sysctl_sched_wakeup_granularity;
    return calc_delta_fair(gran, se);
}
```

如果系统的唤醒抢占太过频繁，大量的上下文切换会影响系统的整体性能。这种情况下可以通过调整 `sched_wakeup_granularity_ns` 的值来解决，`sched_wakeup_granularity_ns` 的值越大，发生唤醒抢占就越不容易。注意 `sched_wakeup_granularity_ns` 的值不要超过调度周期 `sched_latency_ns` 的一半，否则的话，就相当于禁止唤醒抢占了。

^[1] 从几个问题开始理解CFS调度器：<http://linuxperf.com/?p=42>。

5.5 普通进程的组调度

完全公平调度算法会尽力在进程之间保证公平。如果有50个优先级相同的进程，CFS会努力让每个进程获得的CPU时间为2%，以确保公平。

可是考虑一下如图5-20所示的场景。

表面看每个进程都被进程调度器公平对待了，即4个进程每个都获得了25%的CPU时间。但是其中的用户B并没有得到公平的对待。我们将情况考虑得再极端一点：系统上存在50个进程，其中49个都属于用户A，而用户B只有1个进程。那么对于用户B而言，它只能使用2%的CPU资源，这显然是不公平的。

比较合理的做法是，首先确保组间的公平，然后才是组内进程之间的公平，如图5-21所示。

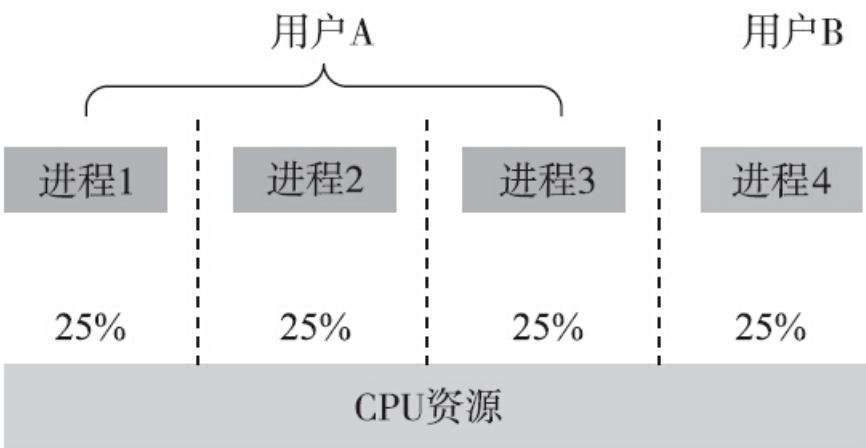


图5-20 没有组调度时的表面公平

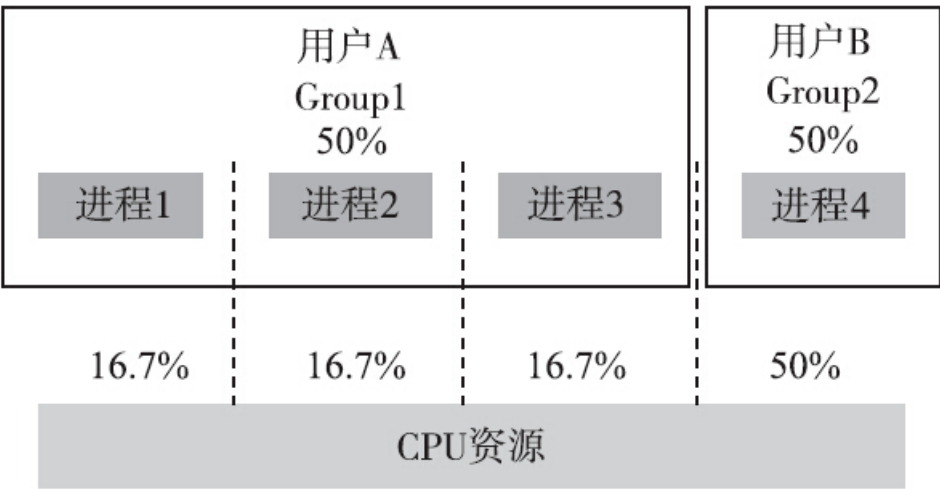


图5-21 引入组调度后

Linux内核实现了cgroups（control groups的缩写）功能，该功能用来限制、记录和隔离一个进程组群所使用的物理资源（如CPU、内存、磁盘IO、网络等）。为了管理不同的资源，cgroups提供了一系

列子系统，本节将要介绍的cpu和后面CPU亲和力一节介绍的cpuset都属于cgroups的子系统。cpu子系统只用于限制进程的CPU使用率。接下来介绍如何使用cgroups的cpu子系统来创建组（task group）及按组来分配CPU资源。

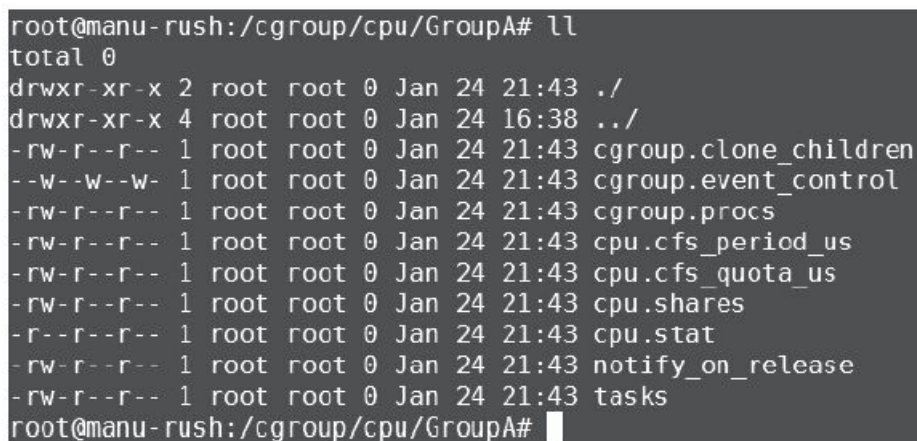
首先需要挂载和安装cgroup文件系统，挂载时需要启用CPU资源控制：

```
mount -t cgroup -o cpu none /cgroup/cpu/
```

在/cgroup/cpu目录下创建两个目录：GroupA和GroupB，这样就会创建两个进程组群。

```
mkdir /cgroup/cpu/GroupA  
mkdir /cgroup/cpu/GroupB
```

目录创建完毕后，可以看到/cgroup/cpu/GroupA和/cgroup/cpu/GroupB目录下都已经存在了很多文件，如图5-22所示。



```
root@manu-rush:/cgroup/cpu/GroupA# ll  
total 0  
drwxr-xr-x 2 root root 0 Jan 24 21:43 ./  
drwxr-xr-x 4 root root 0 Jan 24 16:38 ../  
-rw-r--r-- 1 root root 0 Jan 24 21:43 cgroup.clone_children  
--w--w--w- 1 root root 0 Jan 24 21:43 cgroup.event_control  
-rw-r--r-- 1 root root 0 Jan 24 21:43 cgroup.procs  
-rw-r--r-- 1 root root 0 Jan 24 21:43 cpu.cfs_period_us  
-rw-r--r-- 1 root root 0 Jan 24 21:43 cpu.cfs_quota_us  
-rw-r--r-- 1 root root 0 Jan 24 21:43 cpu.shares  
-r--r--r-- 1 root root 0 Jan 24 21:43 cpu.stat  
-rw-r--r-- 1 root root 0 Jan 24 21:43 notify_on_release  
-rw-r--r-- 1 root root 0 Jan 24 21:43 tasks  
root@manu-rush:/cgroup/cpu/GroupA#
```

图5-22 task group下的配置文件

只需要向GroupA的tasks文件中写入进程ID，该进程就成为了GroupA的成员，当该进程创建子进程时，子进程也会自动成为GroupA中的成员。下面进行一个简单的实验，使用cgroups的cpu子系统来实现分组之间公平地使用CPU资源。

首先打开一个终端，将shell进程的PID写入GroupA的tasks文件中，在该终端上通过stress命令，唤起4个进程执行死循环，消耗CPU资源。

```
echo $$ > /cgroup/cpu/GroupA/tasks  
stress -c 4
```

此时可以看到，/cgroup/cpu/GroupA/tasks中，已经存在多个进程，它们是bash进程和stress进程。其中stress进程有5个：1个几乎不消耗CPU资源的管理进程和4个消耗大量CPU资源的死循环进程。因为该shell中除了stress之外，并无需要消耗CPU的其他进程，所以4个stress进程消耗了几乎所有它能使用的

CPU资源。

同时在另一个终端上，将shell进程的PID写入GroupB，同时通过stress命令，唤起两个进程执行死循环消耗CPU资源，方法如下：

```
echo $$ > /cgroup/cpu/GroupB/tasks
stress -

c 2
```

通过ps命令查看stress相关进程消耗CPU的情况，如图5-23所示。

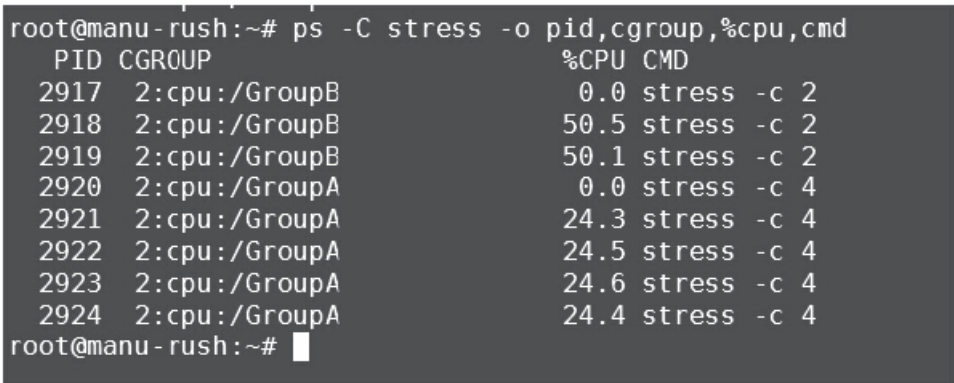


图5-23 GroupA和GroupB下进程的CPU使用情况（1）

可以看到一共有6个消耗CPU的stress进程，共有2个CPU核。平均来说，每个stress进程的使用率应该在33%左右，但是事实并非如此，GroupB的2个进程共消耗了约100%的CPU资源，同时GroupA的4个进程共消耗了约100%的CPU资源。真正做到了兼顾组间公平和组内公平。

在完全公平调度域内，进程有优先级，高优先级的进程享有更多的CPU时间。能否让组与组之间也存在优先级差异，比如GroupB占用的CPU时间是GroupA的2倍？

答案是肯定的。每一个组内都有cpu.shares文件，cpu.shares的值默认是1024，和普通进程默认优先级对应的权重（NICE_0_LOAD）是一样的。这说明进程调度会将整个组看成是1个普通进程来分配CPU资源。

下面调整GroupB的cpu.shares的值，将其调整为2048（GroupA cpu.shares值的两倍），然后重复上面的实验，结果如图5-24所示。

可以看出GroupB中的两个死循环消耗了133%的CPU，而GroupA中的4个死循环只消耗了66%左右的CPU，两者的比例约等于2：1，符合cpu.shares中的比例。



注意 `cpu.shares`的默认值是1024，此时系统将整个组内的所有进程视为一个普通进程。如果系统内存在大量CPU消耗型普通进程，它们不在任何组内，而组内的进程数又很多，那么组内的进程其实处于被损害的地位。此时需要妥善调整`cpu.shares`的值。

```
root@manu-rush:~# ps -C stress -o pid,cgroup,%cpu,cmd
  PID CGROUP          %CPU CMD
  3631 2:cpu:/GroupA      0.0 stress -c 4
  3632 2:cpu:/GroupA     16.6 stress -c 4
  3633 2:cpu:/GroupA     16.6 stress -c 4
  3634 2:cpu:/GroupA     16.6 stress -c 4
  3635 2:cpu:/GroupA     16.6 stress -c 4
  3636 2:cpu:/GroupB       0.0 stress -c 2
  3637 2:cpu:/GroupB     66.3 stress -c 2
  3638 2:cpu:/GroupB     66.4 stress -c 2
root@manu-rush:~#
```

图5-24 GroupA和GroupB下进程的CPU使用情况（2）

5.6 实时进程

对于普通进程来说，完全公平调度已经能够提供足够好的性能和响应体验了。但是某些进程对实时性的要求更高。严格说来实时系统可以分成两类：硬实时进程和软实时进程。

硬实时进程对响应时间的要求非常严格，必须保证在一定的时间内完成，超过时间限制就会失败，而且后果非常严重。这类应用典型的例子有军用武器系统、航空航天系统、交通导航系统、医疗设备等。硬实时的关键特征是任务必须在可保证的时间范围内得到处理。当然这并不意味着所要求的时间范围特别短，而是系统必须保证绝不会超过某一时间范围，无论当时系统的负载如何。主流内核的Linux并不支持硬实时进程，当然有些修改版本提供了该特性。

软实时进程是硬实时的一种弱化形式。尽管软实时进程仍然需要快速响应和要在规定的时间内完成，但是超过了时间的范围也不会有什么灾难性的后果。比较典型的例子是视频处理应用，如果超过了操作时限，则会影响用户体验，但是少量的丢帧还是可以忍受的。

5.6.1 实时调度策略和优先级

Linux提供了两种实时调度的策略：先进先出（SCHED_FIFO）策略和时间片轮转（SCHED_RR）策略。无论进程使用哪种实时策略，其优先级都会高于前面介绍的采用完全公平调度的普通进程。

实时进程也有一个优先级的范围。SUSv3要求至少要为实时策略实现32个离散的优先级。Linux中为实时进程提供了99个实时优先级。从内核层面看，从0到99范围内的优先级属于实时调度范围，从100到139共40个等级属于前面讨论过的完全公平调度的优先级。其中创建普通进程的时候，其优先级的值为完全公平调度中的中间值120。从整体来看，优先级的值越低，其优先级就越高。

事实上每个CPU都有实时运行队列。根据99种离散的优先级可知，共有99个队列。具有相同优先级的实时进程都保存在一个队列之中。这使得在实时调度类中选择下一个运行的进程也就比较简单了，按照优先级从高到低的顺序，选择存在可运行进程的最高优先级队列中的第一个进程即可（如图5-25所示）。事实上内核中还维护有位图来表征哪个优先级的运行队列有可运行的进程，相关结构体定义如下：

```
struct rt_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
    struct list_head queue[MAX_RT_PRIO];
};
struct rt_rq {
    struct rt_prio_array active;
}
```

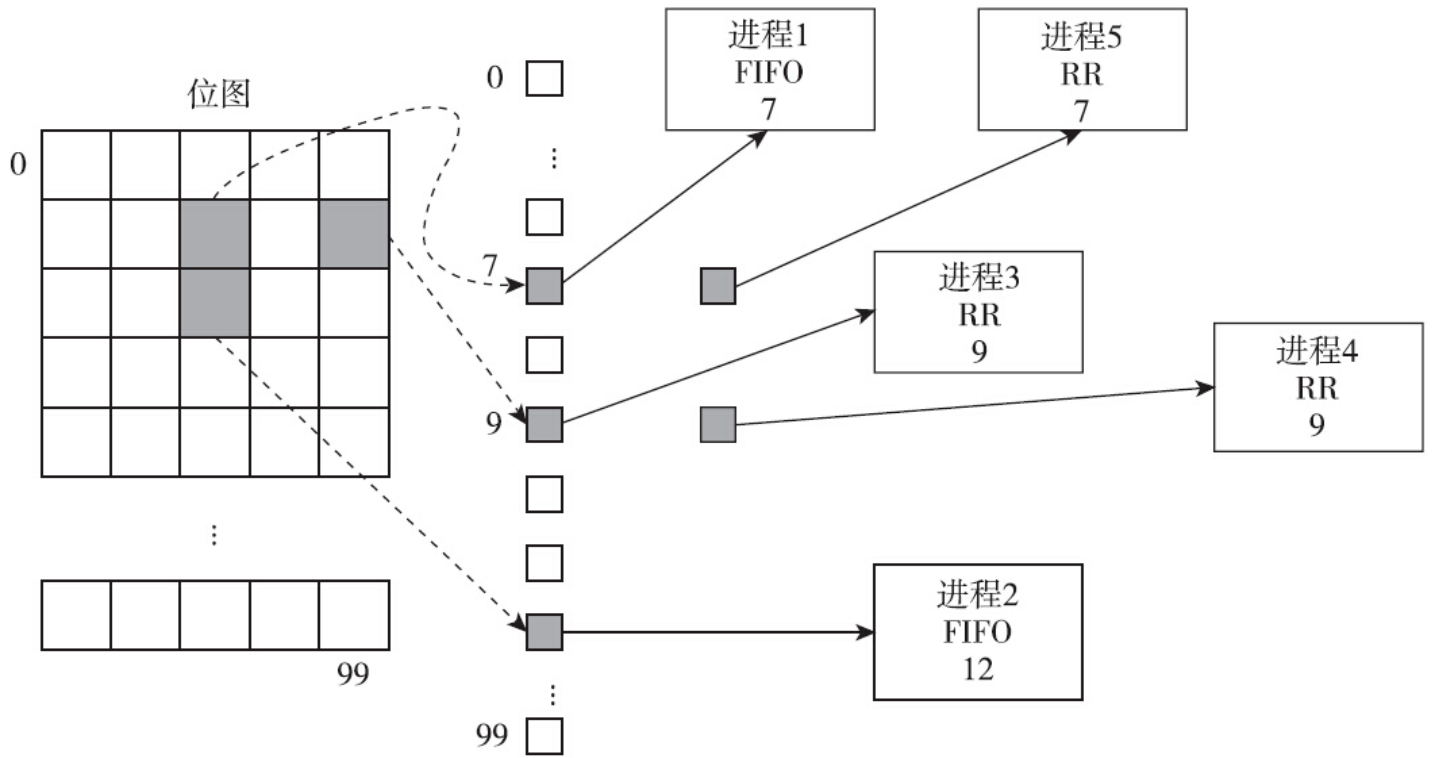


图5-25 实时进程的优先级队列

注意 对于实时进程而言，内核态的优先级和用户进程通过sched_setscheduler或sched_setparam系统调用设置的优先级并不相同：对于内核态而言，优先级的值越小，优先级就越高，而用户进程通过系统调用设置的优先级正好相反，优先级的值越大，优先级越高。两者的换算关系是：

内核态优先级=MAX_RT_PRIO-1-用户态优先级

其中MAX_RT_PRIO的值为100。

1.SCHED_FIFO策略

SCHED_FIFO策略是一种比较简单的策略，即先进先出，它没有时间片的概念，只要没有更高优先级的进程就绪，使用该调度策略的进程就会一直执行。一旦一个调度策略为SCHED_FIFO的进程获得了CPU控制权，它就会始终占有CPU资源直到下面的某种情况发生：

- 自动放弃CPU资源，如执行了一个阻塞型的系统调用或调用了sched_yield系统调用，进程不再处于可执行状态。
- 进程终止了。
- 被一个优先级更高的进程抢占。

如果FIFO类型的进程通过sched_yield系统调用主动让出了CPU，那么内核会将该进程放到对应队列的尾部；如果进程被更高优先级的进程抢占，那么该进程在队列中的位置不变，一旦高优先级的进程停止执行，被抢占的FIFO类型的进程会继续执行。

2.SCHED_RR策略

在时间片轮转的策略中，具有相同优先级的进程轮流执行，进程每次使用CPU的时间为一个固定长度的时间片。使用SCHED_RR策略的实施进程一旦被调度器选中，就会一直占有CPU资源，直到下面的某种情况发生：

- 时间片耗尽。
- 进程自动放弃CPU：或者执行了阻塞式的系统调用，或者主动执行sched_yield函数让出CPU资源。
- 进程终止了。
- 被更高优先级的进程抢占。

前两种情况下，SCHED_RR策略的进程会被放到其优先级运行队列的队尾。最后一种情况下，被抢占的SCHED_RR策略的实施进程仍然位于其运行队列的头部，在更高优先级的进程运行结束后，被抢占的进程会继续执行，直到其时间片的剩余部分耗光为止。

在时间片轮转策略中，时间片的长度是一个关键的参数。POSIX定义了接口来查询SCHED_RR策略的时间片长度：

```
#include <sched.h>
int sched_rr_get_interval(pid_t pid, struct timespec * tp);
```

默认情况下，SCHED_RR类型进程的时间片总是100毫秒。如果内核版本不低于3.9，时间片的大小可以通过调整/proc/sys/kernel/sched_rr_timeslice_ms的值来调整^[1]。

伴随着时钟中断处理程序，scheduler_tick函数会根据当前进程的调度类执行对应的task_tick函数，如下所示：

```
curr->sched_class->task_tick(rq, curr, 0);
```

实时调度类的task_tick函数为task_tick_rt，该函数的实现代码如下所示：

```
static void task_tick_rt(struct rq *rq, struct task_struct*p, int queued)
{
    update_curr_rt(rq);
    watchdog(rq, p);
    /*FIFO类型没有时间片的概念，不会因为执行时间足够长而被抢占

*/
    if (p->policy != SCHED_RR)
        return;
    /*如果时间片还没到，就直接返回

*/
    if (--p->rt.time_slice)
        return; /*时间片已经耗尽，先将进程的时间片重新初始化为默认时间片

*/
    p->rt.time_slice = DEF_TIMESLICE;
    /*如果队列上存在其他进程，则将自身移到队列的尾部，

        *并且设置

need_resched标志位

*/
    if (p->rt.run_list.prev != p->rt.run_list.next) {
        requeue_task_rt(rq, p, 0);
        set_tsk_need_resched(p);
    }
}
```

从上面的代码不难看出，采用SCHED_RR调度策略的实时进程，时间片大小为时钟滴答的整数倍。如果系统CONFIG_HZ为250，那么每4毫秒一个时钟滴答，即时间片大小总是4毫秒的整数倍。

现在的服务器上一般不止一个CPU，在多CPU系统上实时进程的负载均衡是需要解决的问题。严格来讲，对于具有N个CPU的系统，N个最高优先级的可运行状态的实时进程（如果存在大于等于N个实时进程的话）应该占据N个CPU核。对实时进程负载均衡这个话题感兴趣的话，可以阅读《Process Scheduling in Linux》[\[2\]](#)这篇文献，限于篇幅，此话题不再展开论述。

3.SCHED_OTHER策略

SCHED_OTHER策略不属于实时调度的范畴。SCHED_OTHER和下面要讨论的SCHED_BATCH、SCHED_IDLE策略同属于完全公平调度的范畴。事实上，我们遇到的大多数进程都是属于SCHED_OTHER的调度策略。

前面讨论的是nice值在-20~19范围内的进程，都是属于SCHED_OTHER的调度策略。在这种调度策略下，不同的nice值，意味着不同的时间片权重。优先级越高的普通进程，将获得越多的CPU时间。

4.SCHED_BATCH策略

尽管可以通过POSIX实时调度的API设置进程的策略为SCHED_BATCH，但是SCHED_BATCH策略并不属于实时调度的策略。

SCHED_BATCH策略是在Linux 2.6.16的内核中加入的。最初引入这个策略的目的是告知内核，指定这个策略的进程并非交互型的进程，不需要根据休眠时间更改优先级。

这个策略主要用于早期的O(1)调度器，对于完全公平的调度，SCHED_BATCH策略和SCHED_OTHER策略几乎一样。

5.SCHED_IDLE策略

SCHED_IDLE策略也隶属于完全公平调度的范畴。采取SCHED_IDLE调度策略的进程拥有非常低的优先级，比nice值为19的进程的优先级还要低（nice值是19的进程，其权重是15，采用SCHED_IDLE调度策略的进程其权重是3）。一般来说，该策略用于运行优先级非常低的进程，通常在系统中没有其他任务需要使用CPU时这些任务才会运行。

完全公平调度类中负责检查是否应该唤醒抢占的check_preempt_wakeup函数中有如下的语句：

```
if (unlikely(curr->policy == SCHED_IDLE) &&
    likely(p->policy != SCHED_IDLE))
    goto preempt;
```

这段代码表明，在CFS调度域内，如果醒来的候选进程采用的不是SCHED_IDLE策略，而当前运行的进程采用的调度策略是SCHED_IDLE，那么抢占总是会发生。

[1] http://kernelnewbies.org/Linux_3.9。

[2] https://criticalblue.com/news/wp-content/uploads/2013/12/linux_scheduler_notes_final.pdf。

5.6.2 实时调度相关API

Linux下可以通过sched_setscheduler函数来修改进程的调度策略及优先级，其接口定义如下：

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy,
                      const struct sched_param *param);

struct sched_param {
    ...
    int sched_priority;
    ...
}
```

该接口用于修改pid对应进程的调度策略和优先级。当pid等于0时，修改函数调用进程的调度策略和优先级。策略和优先级的有效值如表5-5所示。

表5-5 调度策略

策 略	描 述	sched_param.sched_priority
SCHED_FIFO SCHED_RR	实时进程，先进先出的策略	1～99
	实时进程，时间片轮转的策略	1～99
SCHED_OTHER SCHED_BATCH SCHED_IDLE	普通进程，非实时进程的默认调度策略	0
	普通进程，批处理	0
	比 nice 值为 19 的普通进程优先级还要低	0

sched_setscheduler函数调用成功时返回0，失败时返回-1，并设置errno。

设置进程调度策略和优先级的方法如下面的代码所示。下面的代码将进程的调度策略设置成了SCHED_RR，并且其优先级为99，即实时进程中的最低优先级。

```
struct sched_param sp = { .sched_priority = 99 };
ret = sched_setscheduler(0, SCHED_RR, &sp);
if (ret == -1)
{
    /*error handler*/
}
```

通过fork创建的子进程会保持父进程的调度策略和优先级。有些时候，不希望子进程继承父进程的调度策略和优先级，尤其是父进程是实时进程或nice值是负值的时候。Linux自2.6.32版本开始，提供了SCHED_RESET_ON_FORK选项，一旦设置了该选项，子进程就不会继承父进程的调度策略或nice值了。可通过如下代码设置该标志位：

```
ret = sched_setscheduler(0, SCHED_RR | SCHED_RESET_ON_FORK, &sp);
```

- 如果调用进程的调度策略是SCHED_FIFO或SCHED_RR，那么将fork创建出来的子进程调度策略重设成SCHED_OTHER。
- 如果调用进程的nice值是负值，那么将fork创建出来的进程的nice值重新设置成0。

如何查看进程的调度策略及调度参数？可使用如下语句：

```
int sched_getscheduler(pid_t pid);
int sched_getparam(pid_t pid, struct sched_param *param);
```

sched_getscheduler函数可以返回进程的调度策略，但是无法返回进程的调度参数。

```
int policy = sched_getscheduler(0);
switch(policy)
{case SCHED_OTHER:    /**/
 case SCHED_FIFO:    /**/
 case SCHED_RR:      /**/
    ...
}
```

对于实时进程，可以调用sched_getparam函数来获得其优先级，代码如下所示：

```
struct sched_param sp;

int ret ;

ret = sched_getparam(0,&sp);
if(ret == -1)
{ /*error handle here*/
}
printf("

process priority is %d\n"

,sp.sched_priority);
```

`sched_setscheduler`函数用来同时设置调度策略和调度参数。除了该接口外，Linux还提供了一个功能弱化的函数即`sched_setparam`，该函数可以用来调整进程的调度参数，定义如下：

```
#include <sched.h>
int sched_setparam(pid_t pid, const struct sched_param *param);
```

通过该接口，可以调整实时进程的优先级，使用方法如下：

```
struct sched_param sp ;
sp.sched_priority = 15;
ret = sched_setparam(0,&sp);
if(ret == -1)
{
    /*error handler*/
}
```

可以通过`ps`命令的输出来看进程的调度策略和优先级：

```
manu@manu-rush:~$ ps -p 7110 -o pid,cmd,sched,rtprio,pri
  PID CMD          SCH RTPRIO PRI
  7110 sleep 100      2      99 139
```

在`sched`字段中`SCHED_OTHER`、`SCHED_FIFO`、`SCHED_RR`、`SCHED_BATCH`和`SCHED_IDLE`对应的值分别为0、1、2、3和5。

除了`ps`命令外，`util-linux`包中提供了`chrt`工具，可以查看和修改进程的调度策略和优先级。

查看进程的调度策略和优先级的方法如下：

```
manu@manu-rush:~$ chrt -p 7125
pid 7125's current scheduling policy: SCHED_RR
pid 7125's current scheduling priority: 77
manu@manu-rush:~$ chrt -p 1
pid 1's current scheduling policy: SCHED_OTHER
pid 1's current scheduling priority: 0
```

修改进程的调度策略和优先级的方法如下：

```
/*7135进程最初是普通进程，调度策略为

SCHED_OTHER*/
root@manu-rush:~# chrt -p 7135
pid 7135's current scheduling policy: SCHED_OTHER
pid 7135's current scheduling priority: 0
/*-r表示修改调度策略为

SCHED_RR.

40表示修改优先级为

40*/
root@manu-rush:~# chrt -p -r 40 7135
root@manu-rush:~# chrt -p 7135
pid 7135's current scheduling policy: SCHED_RR
pid 7135's current scheduling priority: 40
/*-f表示修改调度策略为
```

SCHED_FIFO.

20表示修改优先级为

```
20*/
root@manu-rush:~# chrt -p -f 20 7135
root@manu-rush:~# chrt -p 7135
pid 7135's current scheduling policy: SCHED_FIFO
pid 7135's current scheduling priority: 20
```

5.6.3 限制实时进程运行时间

实时进程的优先级高于普通进程，如果实时进程处于可执行的状态，那么普通进程无法获得CPU资源。如果使用实时调度策略的进程出现了bug，始终处于可运行的状态，系统将不会调度其他普通进程。这种情况是非常危险的，系统很可能会失去控制，而用户甚至超级用户也无能为力。

为了防止出现这种情况，系统做了改进，纵然始终存在可以运行的实时进程，仍然允许普通进程获得一定的CPU时间。

系统提供了控制选项来控制单位时间内最多分配多少CPU时间给实时进程。在Linux中，这两个控制参数为：

```
sysctl -n kernel.sched_rt_period_us
1000000
sysctl -n kernel.sched_rt_runtime_us
950000
```

这两个参数的含义是在以`sched_rt_period_us`为一个周期的时间内，所有实时进程运行的时间总和不超过`sched_rt_runtime_us`。这两个配置项的默认值为1秒和0.95秒，表示每秒钟为一个周期，所有实时进程运行的总时间不超过0.95秒，剩下的0.05秒留给普通进程。有了这个机制，哪怕始终有实时进程处于TASK_RUNNING状态，普通进程也能获得运行的机会。

如果在一个周期的时间内，实时进程对CPU的需求不足0.95秒，那么剩余的时间都会分配给普通的进程。而如果实时进程对CPU的需求大于0.95秒，它也只能够运行0.95秒，剩下的0.05秒留给其他普通进程。但是如果0.05秒内并没有任何普通进程处于可运行状态，实时进程能否运行超过0.95秒吗？答案还是不能，内核宁可让CPU闲着，也不给实时进程使用。

但是前面讨论的场景都是单CPU的场景，如果存在N个CPU，那么所有CPU上的所有实时进程占有CPU的上限应该为 $N * \text{sched_rt_runtime_us} / \text{sched_rt_period_us}$ 。有的CPU上实时进程对CPU的需求超过`sched_rt_runtime_us`，而有的CPU上实时进程对CPU的需求不足`sched_rt_runtime_us`，因此内核允许CPU之间互相拆借。若实时进程在CPU上占用的时间超过了`sched_rt_runtime_us`，则该实时进程会尝试去其他CPU上借时间，将其他CPU剩余的时间借过来。这样做的好处是避免了进程在CPU之间迁移导致的上下文切换、缓存失效等开销。这部分逻辑出现在`kernel/sched_rt.c`中的`sched_rt_runtime_exceeded`函数，该函数会通过`balance_runtime`函数向其他CPU借用时间 [1]。

事实上，实时进程也支持组调度，可以控制一组实时进程（`task_group`）占用的CPU时间，将CPU占用的管理分配得更加细致 [2]。

[1] Linux进程组调度机制分析: <http://www.oenhan.com/task-group-sched#toc-4>。

[2] Linux组调度浅析: http://kouucocu.lofter.com/post/1cdb8c4b_50f6314。

5.7 CPU的亲合力

在对称多处理器（SMP）环境中，一个进程被重新调度时，不一定是在上次执行的CPU上运行。

同一个进程在不同CPU之间迁移会带来性能的损失，损失的主要原因在于缓存。在进程迁移到新的处理器上后写入新数据到内存时，原有处理器的缓存就过期了。当进程在不同处理器之间迁移时，会带来两方面的性能损失：

- 进程不能访问老的缓存数据；
- 原处理器中缓存中的数据必须标记为无效。

由于迁移会带来性能损失，因此进程调度器趋于把进程固定在一个处理器上执行。

如何查看进程当前运行在哪个CPU上？可以通过ps命令的PSR字段来查看进程当前执行或上一次执行时所在的CPU编号。因为进程调度并不保证进程总是固定在某个CPU上，所以多次查看进程的PSR，其值可能会发生变化。

```
root@manu-rush:~# ps -p 7214 -o pid,cmd,psr
PID CMD          PSR
7214 sleep 1000    0
```

有时候需要把进程绑定到某个或某几个CPU上运行。这就需要设置进程的CPU硬亲和力了。Linux提供了非标准的系统调用来获取和修改进程的硬亲和力：即sched_setaffinity函数和sched_getaffinity函数。

sched_setaffinity函数用来设置pid指定进程的CPU亲和力，如果pid的值为0，那么该函数用来修改调用进程的CPU亲和力。函数接口定义如下：

```
#define _GNU_SOURCE
#include <sched.h>
int sched_setaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);
```

cpu_set_t数据结构是位掩码，但是不应该直接操作cpu_set_t类型的变量。Linux提供了一组宏来操作cpu_set_t类型的变量：

```
/*将

set初始化为空

*/
void CPU_ZERO(cpu_set_t *set);
/*将

cpu指定的

CPU添加到

set中

*/
void CPU_SET(int cpu, cpu_set_t *set);
/*从

set中删除

CPU cpu*/
void CPU_CLR(int cpu, cpu_set_t *set);
/*判断

CPU cpu是否
```

set中的成员

```
*/
int CPU_ISSET(int cpu, cpu_set_t *set);
```

CPU集合中的编号从0开始。一般在调用CPU_XXX系列函数之前，需要对系统中的CPU核数了然于胸，才能有的放矢。指定cpu的值比系统中的最大CPU编号还大是没有意义的。nproc命令和lscpu命令都可以获取系统的CPU核数，代码如下：

```
manu@manu-rush:~$ nproc
2
manu@manu-rush:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
```

通过proc文件系统的/proc/cpuinfo也可以获取CPU的核数。

可以通过下面的代码将某进程迁移到CPU 1上：

```
cpu_set_t set;
/*必须首先调用
```

CPU_ZERO清空，不可想当然地认为是空

```
*/
CPU_ZERO(&set);
CPU_SET(1, &set);
sched_setaffinity(pid, sizeof(cpu_set_t), &set);
```

Linux提供了sched_getaffinity接口来查看进程的CPU亲和力：

```
cpu_set_t set;
/*必须首先调用
```

CPU_ZERO清空，不可想当然地认为是空

```
*/
CPU_ZERO(&set);
CPU_SET(1, &set);
sched_setaffinity(pid, sizeof(cpu_set_t), &set);
```

调用sched_getaffinity之前，需要先调用CPU_ZERO将set清空。函数调用成功时，会将结果记录在set中，但是不要直接操作set来判断哪些CPU在集合中，而是应该用CPU_ISSET来判断。

内核如何保证进程只会在某些CPU上执行？内核中的进程对应的进程描述符中有个cpumask_t类型的成员变量cpus_allowed，该成员变量会记住进程允许的CPU。内核在调度的时候会通过select_task_rq来选择CPU，只会选择出允许的CPU。

```
static inline
int select_task_rq(struct task_struct *p, int sd_flags, int wake_flags)
{
    int cpu = p->sched_class->select_task_rq(p, sd_flags, wake_flags);
    if (unlikely(!cpumask_test_cpu(cpu, tsk_cpus_allowed(p)) ||
                !cpu_online(cpu)))
        cpu = select_fallback_rq(task_cpu(p), p);
    return cpu;
}
```

有个很有意思的话题是内核调用select_task_rq的时机：当新的进程创建出来时，当进程调用exec时，当进程从睡眠中醒来时，都是调用select_task_rq的好时机（如图5-26所示），可以通过这些时机来实现各个CPU之间的负载均衡。

除了编程接口可以获取和修改进程的亲和力以外，Linux的util-linux包中还提供了taskset工具以命令行的方式做同样的事情。它查询进程的CPU亲和力的方法如下：

```
manu@manu-rush:~$ taskset -p 1
pid 1's current affinity mask: 3
```

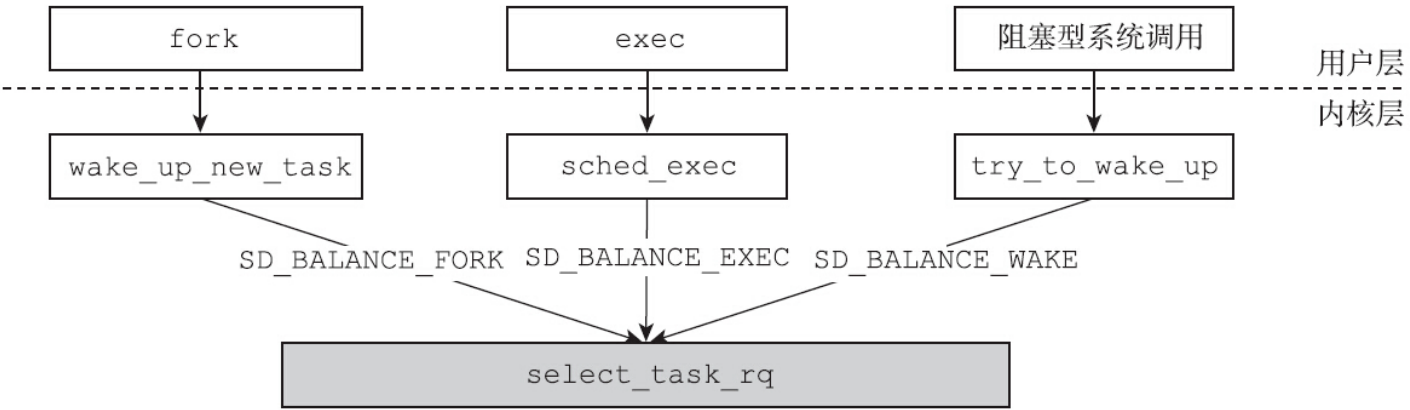


图5-26 调用select_task_rq的时机

进程1的mask为3=0x11，即允许在CPU 0和CPU 1上运行。

修改进程的CPU亲和力的方法如下：

```
/*允许进程

700运行在

CPU0,CPU3,CPU7,CPU8,CPU9,CPU10,CPU11上

*/
taskset -pc 0,3,7-11 700
manu@manu-rush:~$ sudo taskset -pc 1 2000
pid 2000's current affinity list: 0,1
pid 2000's new affinity list: 1
```

除了taskset工具外，cpuset也可以用来设定CPU亲和力。cpuset是Linux控制组（control groups）中的一个子系统，该子系统的用途是管理进程可以使用的CPU核心和内存节点。使用cpuset之前，首先要确认内核是否已经提供了对cpuset功能的支持：

```
root@manu-rush:/boot# grep "CONFIG_CPUSETS" config-3.13.0-32-generic
CONFIG_CPUSETS=y
```

Linux将cgroups实现成了文件系统。在较新的Linux发行版本中（如Ubuntu 14.04），系统已经挂载了所有的cgroup子系统，通过mount-t cgroup命令来查看。如果操作系统并没有挂载cpuset子系统，那么可以通过如下命令手工挂载：

```
mkdir /dev/cpuset
mount -t cgroup -o cpuset none /dev/cpuset
```

执行完挂载后，通过mount命令可以看到一个新的cpuset类型的文件系统挂载到了/dev/cpuset目录下：

```
none on /dev/cpuset type cgroup (rw,cpuset)
```

我们可以创建一个新的CPU分配组，比如GroupA，方法很简单，就是在/cgroup下创建一个目录，方法如下：

```
mkdir /dev/cpuset/GroupA
```

在cpuset中设置了新的群组之后，该群组的目录下有很多的文件，对应不同的配置项，如图5-27所示。大部分配置项都是可选的，但cpuset.cpus和cpuset.mems这两项分别用来指定群组允许使用的CPU核心和内存节点，是强制配置项，必须要指定。


```
root@manu-rush:/dev/cpuset/GroupA# ll
total 0
drwxr-xr-x 2 root root 0 Jan 24 23:55 ./
drwxr-xr-x 3 root root 0 Jan 24 23:54 ../
-rw-r--r-- 1 root root 0 Jan 24 23:55 cgroup.clone_children
--w--w--w- 1 root root 0 Jan 24 23:55 cgroup.event_control
-rw-r--r-- 1 root root 0 Jan 24 23:55 cgroup.procs
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.cpu_exclusive
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.cpus
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.mem_exclusive
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.mem_hardwall
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.memory_migrate
-r--r--r-- 1 root root 0 Jan 24 23:55 cpuset.memory_pressure
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.memory_spread_page
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.memory_spread_slab
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.mems
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.sched_load_balance
-rw-r--r-- 1 root root 0 Jan 24 23:55 cpuset.sched_relax_domain_level
-rw-r--r-- 1 root root 0 Jan 24 23:55 notify_on_release
-rw-r--r-- 1 root root 0 Jan 24 23:55 tasks
root@manu-rush:/dev/cpuset/GroupA#
```

图5-27 cpuset子系统下task group的配置文件

下面我们通过如下语句将GroupA中所有的进程限制在CPU 1上：

```
echo "1" > /dev/cpuset/GroupA/cpuset.cpus
echo "0" > /dev/cpuset/GroupA/cpuset.mems
```

设置好GroupA允许使用的CPU后，就可以将某些进程放入GroupA中了，按照设定，这些进程只会使用CPU 1。比如将shell本身的PID归于GroupA，这样在该shell上启动的所有进程都会归于这个GroupA下：

```
echo $$ > /dev/cpuset/GroupA/tasks
```

在该shell上通过stress-c 4命令，启动4个进程执行死循环，消耗大量的CPU资源，通过ps命令可以看到，这4个进程总是运行在CPU 1上，如图5-28所示。

```
manu@manu-rush:~$ ps -C stress -o pid,psr,cmd,etime,cgroup,%cpu
  PID PSR CMD                                ELAPSED CGROUP                                %CPU
  3795  1 stress -c 4                        01:25  2:cpuset:/GroupA                            0.0
  3796  1 stress -c 4                        01:25  2:cpuset:/GroupA                            24.9
  3797  1 stress -c 4                        01:25  2:cpuset:/GroupA                            24.9
  3798  1 stress -c 4                        01:25  2:cpuset:/GroupA                            24.9
  3799  1 stress -c 4                        01:25  2:cpuset:/GroupA                            24.9
manu@manu-rush:~$
```

图5-28 通过cpuset绑定到CPU 1上运行的进程ps的输出

第6章 信号

信号是一种软件中断，用来处理异步事件。内核递送这些异步事件到某个进程，告诉进程某个特殊事件发生了。这些异步事件，可能来自硬件，比如访问了非法的内存地址，或者除以0了；可能来自用户的输入，比如shell终端上用户在键盘上敲击了Ctrl+C；还可能来自另一个进程，甚至有些来自进程自身。

信号的本质是一种进程间的通信，一个进程向另一个进程发送信号，内核至少传递了信号值这个字段。实际上，通信的内容不止是信号值。

信号机制是Unix家族里一个古老的通信机制。传统的信号机制有一些弊端，更为严重的是，信号处理函数的执行流和正常的执行流同时存在，给编程带来了很多的麻烦和困扰，一不小心就可能掉入陷阱。本章将会介绍信号的方方面面，包括传统信号的弊端，Linux对信号机制的改进，以及信号机制里面的陷阱，希望对读者能有所帮助。

6.1 信号的完整生命周期

前文提到过，信号的本质是一种进程间的通信。进程之间约定好：如果发生了某件事情 T（trigger），就向目标进程（destination process）发送某特定信号 X，而目标进程看到 X，就意识到 T 事件发生了，目标进程就会执行相应的动作 A（action）。

接下来以配置文件改变为例，来描述整个过程。很多应用都有配置文件，如果配置文件发生改变，需要通知进程重新加载配置。一般而言，程序会默认采用 SIGHUP 信号来通知目标进程重新加载配置文件。

目标进程首先约定，只要收到 SIGHUP，就执行重新加载配置文件的动作。这个行为称为信号的安装（installation），或者信号处理函数的注册。安装好了之后，因为信号是异步事件，不知道何时会发生，所以目标进程依然正常地干自己的事情。某年某月的某一天，管理员突然改变了配置文件，想通知这个目标进程，于是就向目标进程发送了信号。他可能在终端执行了 kill-SIGHUP 命令，也可能调用了 C 的 API，不管怎样，信号产生了。这时候，Linux 内核收到了产生的信号，然后就在目标进程的进程描述符里记录了一笔：收到信号 SIGHUP 一枚。Linux 内核会在适当的时机，将信号递送（deliver）给进程。在内核收到信号，但是还没有递送给目标进程的这一段时间里，信号处于挂起状态，被称为挂起（pending）信号，也称为未决信号。内核将信号递送给进程，进程就会暂停当前的控制流，转而去执行信号处理函数。这就是一个信号的完整生命周期。

一个典型的信号会按照上面所述的流程来处理，但是实际情况要复杂得多，还有很多场景需要考虑，比如：

- 目标进程正在执行关键代码，不能被信号中断，需要阻塞某些信号，那么在这期间，信号就不允许被递送到进程，直到目标进程解除阻塞。

- 内核发现同一个信号已经存在，那么它该如何处理这种重复的信号，排队还是丢弃？

- 内核递送信号的时候，发现已有多个不同的信号被挂起，那它应该优先递送哪个信号？

- 对于多线程的进程，如果向该进程发送信号，应该由哪个线程来负责响应？

这些问题，在接下来的章节中会逐一得到解决。

6.2 信号的产生

作为进程间通信的一种手段，进程之间可以互相发送信号，然而发给进程的信号，通常源于内核，包括：

- 硬件异常。
- 终端相关的信号。
- 软件事件相关的信号。

6.2.1 硬件异常

硬件检测到了错误并通知内核，由内核发送相应的信号给相关进程。和硬件异常相关的信号见表6-1。

表6-1 与硬件异常有关的信号

信 号	值	说 明
SIGBUS	7	总线错误，表示发生了内存访问错误
SIGFPE	8	表示发生了算术错误，尽管 FPE 是浮点异常的缩写
SIGILL	9	进程尝试执行非法的机器语言指令
SIGSEGV	11	段错误，表示应用程序访问了无效地址

常见的能触发SIGBUS信号的场景有：

·变量地址未对齐：很多架构访问数据时有对齐的要求。比如int型变量占用4个字节，因此架构要求int变量的地址必须为4字节对齐，否则就会触发SIGBUS信号。

·mmap映射文件：使用mmap将文件映射入内存，如果文件大小被其他进程截短，那么在访问文件大小以外的内存时，会触发SIGBUS信号。

虽然SIGFPE的后缀FPE是浮点异常（Float Point Exception）的含义，但是该异常并不限于浮点运算，常见的算术运算错误也会引发SIGFPE信号。最常见的就是“整数除以0”的例子。

SIGILL的含义是非法指令（illegal instruction）。一般表示进程执行了错误的机器指令。下面来看一段示例代码：

```
typedef void(*FUNC)(void);
int main(void)
{
    const static unsigned char insn[4] = { 0xff, 0xff, 0xff, 0xff };
    FUNC function = (FUNC) insn;
    function();
}
```

上述代码中，因为函数地址不是合法有效的值，所以触发了SIGILL错误。发生这种错误，一般是函数指针遭到破坏，当执行函数指针指向的函数时，就会触发SIGILL信号。另外也可能是由指令集的演进引起的。比如，很多在新的体系结构中编译出来的可执行程序，在老的机器上可能会无法运行，故而在老机器上运行时，也可能产生SIGILL信号。

SIGSEGV是所有C程序员的噩梦。没经历几个刻骨铭心的段错误，很难成长为合格的C程序员。由于C语言可以直接操作指针，就像时常行走在河边的顽童很难避免湿鞋一样，程序员很难避免段错误，没有经验的程序员更是如此。常见的情况有：

- 访问未初始化的指针或NULL指针指向的地址。
- 进程企图在用户态访问内核部分的地址。
- 进程尝试去修改只读的内存地址。

当然，程序员不会直接去做这种傻事，一般来说是由于程序的错误，导致原本存放的指针被篡改成错乱值，因而在访问指针指向的变量时，触发了SIGSEGV信号。

前面所讲的这四种硬件异常，一般是由程序自身引发的，不是由其他进程发送的信号引发的，并且这些异常都比较致命，以至于进程无法继续下去。所以这些信号产生之后，会立刻递送给进程。默认情况下，这四种信号都会使进程终止，并且产生core dump文件以供调试。对于这些信号，进程既不能忽略，也不能阻塞。

6.2.2 终端相关的信号

对于Linux程序员而言，终端操作是免不了的。终端有很多的设置，可以通过执行如下指令来看：

```
stty -a
```

很重要的是，终端定义了如下几种信号生成字符：

·Ctrl+C：产生SIGINT信号。

·Ctrl+\：产生SIGQUIT信号。

·Ctrl+Z：产生SIGTSTP信号。

键入这些信号生成字符，相当于向前台进程组发送了对应的信号。

另一个和终端关系比较密切的信号是SIGHUP信号。很多程序员都遇到过这种问题：使用ssh登录到远程的Linux服务器，执行比较耗时的操作（如编译项目代码），却因为网络不稳定，或者需要关机回家，ssh连接被断开，最终导致操作中途被放弃而失败。

之所以会如此，是因为一个控制进程在失去其终端之后，内核会负责向其发送一个SIGHUP信号。在登录会话中，shell通常是终端的控制进程，控制进程收到SIGHUP信号后，会引发如下的连锁反应。

shell收到SIGHUP后会终止，但是在终止之前，会向由shell创建的前台进程组和后台进程组发送SIGHUP信号，为了防止处于停止状态的任务接收不到SIGHUP信号，通常会在SIGHUP信号之后，发送SIGCONT信号，唤醒处于停止状态的任务。前台进程组和后台进程组的进程收到SIGHUP信号，默认的行为是终止进程，这也是前面提到的耗时任务会中途失败的原因。

注意，单纯地将命令放入后台执行（通过&符号，如下所示），并不能摆脱被SIGHUP信号追杀的命运。

```
command &
```

那么如何让进程在后台稳定地执行而不受终端连接断开的影响呢？可以采用如下方法。

1.nohup

可以使用如下方式执行命令：

标准输入会重定向到/dev/null，标准输出和标准错误会重定向到nohup.out，如果无权限写入当前目录下的nohup.out，则会写入home目录下的nohup.out。

2.setsid

使用如下方式执行命令：

这种方式 and nohup 的原理不太一样。nohup 仅仅是使启动的进程不再响应 SIGHUP 信号，但是 setsid 则完全不属于 shell 所在的会话了，并且其父进程也已经不是 shell 而是 init 进程了。

```
manu@manu-hacks:~$ nohup sleep 200 &
[1] 11686
nohup: 忽略输入并把输出追加到

"nohup.out"
manu@manu-hacks:~$ ps -o cmd,pid,ppid,pgid,sid,etime
CMD          PID  PPID  PGID  SID    ELAPSED
-bash        11365 11364 11365 11365   03:59
sleep 200    11686 11365 11686 11365   00:30
ps -o cmd,pid,ppid,pgid,sid 11750 11365 11750 11365   00:00
manu@manu-hacks:~$ setsid sleep 300 &
[1] 11910
[1]+  已完成

          setsid sleep 300
manu@manu-hacks:~$ ps -p 11912 -o cmd,pid,ppid,pgid,sid,etime
CMD          PID  PPID  PGID  SID    ELAPSED
sleep 300    11912  1 11912 11912   00:48
```

3.disown

很多情况下，启动命令时，忘记了使用nohup或setsid，可还有办法亡羊补牢？

答案是使用作业控制里面的disown，方法如下：

```
manu@manu-hacks:~$ sleep 1004 &
[1] 13861
manu@manu-hacks:~$ jobs -l
[1]+ 13861 运行中
```

```
          sleep 1004 &
manu@manu-hacks:~$ disown %1
manu@manu-hacks:~$ jobs -
```

```
1
manu@manu-hacks:~$ exit
```

使用disown之后，shell退出时，就不会向这些进程发送SIGHUP信号了。在另一个终端上，仍然可以看到sleep 1004在运行：

```
manu@manu-hacks:~$ ps -ef|grep sleep
manu      13861      1  0 12:42 ?        00:00:00 sleep 1004
```

当然，还有其他的方法可以做到这点，如screen命名。对这个感兴趣的朋友可以阅读网上的参考资料 [\[1\]](#)。

[\[1\]](#) Linux技巧：让进程在后台可靠执行的几种方法，<https://www.ibm.com/developerworks/cn/linux/l-cn-nohup/>。

6.2.3 软件事件相关的信号

软件事件触发信号产生的情况也比较多：

- 子进程退出，内核可能会向父进程发送SIGCHLD信号。
- 父进程退出，内核可能会给子进程发送信号。
- 定时器到期，给进程发送信号。

我们已熟知子进程退出时会向父进程发送SIGCHLD信号。这点在第4章中已经做了很详细的分析。

与子进程退出向父进程发送信号相反，有时候，进程希望父进程退出时向自己发送信号，从而可以得知父进程的退出事件。Linux也提供了这种机制。

每一个进程的进程描述符task_struct中都存在如下成员变量：

```
int pdeath_signal; /* The signal sent when the parent dies */
```

如果父进程退出，子进程希望收到通知，那么子进程可以通过执行如下代码来做到：

```
prctl(PR_SET_PDEATHSIG, sig);
```

父进程退出时，会遍历其子进程，发现有子进程很关心自己的退出，就会向该子进程发送子进程希望收到的信号。

很多定时器相关的函数，背后都牵扯到信号，具体见表6-2。

表6-2 定时器相关的信号

函 数	相 关 信 号
alarm	SIGALRM (14)
ualarm	SIGALRM (14)
setitimer: ITIMER_REAL	SIGALRM (14)
setitimer: ITIMER_VIRTUAL	SIGVTALRM (26)
setitimer: ITIMER_PROF	SIGPROF (27)
timer_create	可以由用户来指定

6.3 信号的默认处理函数

从上一节可以看出，信号产生的源头有很多。那么内核将信号递送给进程后，进程会执行什么操作呢？

很多信号尤其是传统的信号，都会有默认的信号处理方式。如果我们不改变信号的处理函数，那么收到信号之后，就会执行默认的操作。

信号的默认操作有以下几种：

- 显式地忽略信号：即内核将会丢弃该信号，信号不会对目标进程产生任何影响。
- 终止进程：很多信号的默认处理是终止进程，即将进程杀死。
- 生成核心转储文件并终止进程：进程被杀死，并且产生核心转储文件。核心转储文件记录了进程死亡现场的信息。用户可以使用核心转储文件来调试，分析进程死亡的原因。
- 停止进程：停止进程不同于终止进程，终止进程是进程已经死亡，但是停止进程仅仅是使进程暂停，将进程的状态设置成 TASK_STOPPED，一旦收到恢复执行的信号，进程还可以继续执行。
- 恢复进程的执行：和停止进程相对应，某些信号可以使进程恢复执行。

这5种行为的简单标记如下：

- ignore
- terminate
- core
- stop
- continue

事实上，根据信号的默认操作，可以将传统信号分成5派，具体见表6-3到表6-7。

表6-3 ignore派的信号

信 号	值	说 明
SIGCHLD	17	子进程终止、停止或恢复执行
SIGURG	23	套接字上的紧急数据
SIGWINCH	28	终端窗口大小发生变化

表6-4 terminate派的信号

信 号	值	说 明
SIGHUP	1	挂起 (hangup), 多用于终端断开
SIGINT	2	终端中断
SIGKILL	9	杀死进程, 该信号不能被忽略, 不能被屏蔽, 用户不能将信号处理函数改写成用户定义的函数
SIGUSR1	10	用户自定义信号 1
SIGUSR2	12	用户自定义信号 2
SIGPIPE	13	管道断开, 多见于 socket 通信
SIGALRM	14	定时器到期, 该信号多用于实现定时器
SIGTERM	15	终止进程。因为 SIGKILL 过于残暴, 进程终止时, 可能需要先执行一些操作来保存现场信息, 所以合理地杀死进程的方法是先发送 SIGTERM 信号, 稍等片刻, 再发送 SIGKILL 信号
SIGSTKFLT	16	协处理器栈错误, Linux 并未使用该信号
SIGVTALRM	26	虚拟定时器过期, setitimer 函数的 ITIMER_VIRTUAL 模式
SIGPROF	27	性能分析定时器过期, setitimer 函数的 ITIMER_PROF 模式
SIGIO	29	I/O 时可能发生
SIGPWR	30	电量将要耗尽

表6-5 core派的系统调用

信 号	值	说 明
SIGQUIT	3	终端 Ctrl+\ 可产生该信号
SIGILL	4	非法的指令
SIGTRAP	5	跟踪 / 断点陷阱, gdb/strace 一类工具会使用该信号 ^① 。这类工具会拦截或修改 SIGTRAP 信号的信号处理函数

(续)

信 号	值	说 明
SIGABRT	6	进程中止, 进程调用 abort 函数会向自身发送 SIGABRT 信号, 此外如果使用了断言 assert, assert 失败时也会产生 SIGABRT 信号
SIGBUS	7	总线错误
SIGFPE	8	算术异常
SIGSEGV	11	段错误, 访问了非法的地址
SIGXCPU	24	突破了对 CPU 时间的限制
SIGXFSZ	25	突破了对文件大小的限制
SIGSYS	31	无效的系统调用

(注: ptrace/SIGTRAP/int3的关联, http://blog.linux.org.tw/~jserv/archives/2010/08/ptrace_sigtrap.html。)

表6-6 stop派的信号

信 号	值	说 明
SIGSTOP	19	确保进程会停止，该信号不能被忽略，不能将信号处理函数改写成用户指定的函数
SIGTSTP	20	终端停止信号，和 SIGSTOP 功能类似，但是可以被进程忽略，可以被捕捉执行用户指定的信号处理函数
SIGTTIN	21	用于作业控制，如果后台进程组尝试对终端执行 read 操作，终端驱动程序就会向该进程组发送 SIGTTIN 信号
SIGTTOU	22	如果终端启用了 TOSTOP（如通过 stty tostop 命令）即不允许后台进程向终端写入，而某一后台进程尝试写入终端时，终端驱动程序就会向进程组发送 SIGTTOU 信号

表6-7 continue派的信号

信 号	值	说 明
SIGCONT	18	如果目标进程处于停止状态，则恢复执行

信号的这些默认行为是非常有用的。比如停止行为和恢复执行。系统可能有一些备份的工作，这些工作优先级并不高，但是却消耗了大量的I/O资源，甚至是CPU资源（比如需要先压缩再备份）。这样的工作一般是在夜深人静，业务稀少的时候进行的。在业务比较繁忙的情况下，如果备份工作还在进行，则可能会影响到业务。这时候停止和恢复就非常有用。在业务繁忙之前，可以通过SIGSTOP信号将备份进程暂停，在几乎没有什么业务的时候，通过SIGCONT信号使备份进程恢复执行。

很多信号产生核心转储文件也是非常有意义的。一般而言，程序出错才会导致SIGSEGV、SIGBUS、SIGFPE、SIGILL及SIGABRT等信号的产生。生成的核心转储文件保留了进程死亡的现场，提供了大量的信息供程序员调试、分析错误产生的原因。核心转储文件的作用有点类似于航空中的黑盒子，可以帮助程序员还原事故现场，找到程序漏洞。

很多情况下，默认的信号处理函数，可能并不能满足实际的需要，这时需要修改信号的信号处理函数。信号发生时，不执行默认的信号处理函数，改而执行用户自定义的信号处理函数。为信号指定新的信号处理函数的动作，被称为信号的安装。glibc提供了signal函数和sigaction函数来完成信号的安装。signal出现得比较早，接口也比较简单，sigaction则提供了精确的控制。

6.4 信号的分类

在Linux的shell终端，执行kill-l，可以看到所有的信号：

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

这些信号可以分成两类：

- 可靠信号。
- 不可靠信号。

信号值在[1， 31]之间的所有信号，都被称为不可靠信号；在[SIGRTMIN， SIGRTMAX]之间的信号，被称为可靠信号。

不可靠信号是从传统的Unix继承而来的。早期Unix系统信号的机制并不完备，在实践过程中暴露了很多弊端，因此把这些早期出现的信号值在[1， 31]之间的信号称之为不可靠信号。所谓不可靠，指的是发送的信号，内核不一定能递送给目标进程，信号可能会丢失。

随着时间的流逝，人们意识到原有的信号机制存在弊端。但是[1， 31]之间的信号存在已久，在很多应用中被广泛使用，出于兼容性的考虑，不能改变这些信号的行为模式，所以只能新增信号。新增的信号就是我们今天看到的在[SIGRTMIN， SIGRTMAX]范围内的信号，它们被称为可靠信号。

对信号有了初步了解后，知道signal和sigaction函数接口的读者可能会产生误解，认为用signal函数安装、用kill函数（或者tkill函数）发送的信号，就是不可靠信号；用sigaction函数安装、用sigqueue函数发送的信号，就是可靠信号。这种理解是错误的。信号的可靠与否，完全取决于信号的值，而与采用哪种方式安装或发送无关。

说了这么多，不可靠信号和可靠信号的根本差异到底在哪里？根本差异在于收到信号后，内核有不同的处理方式。

对于不可靠信号，内核用位图来记录该信号是否处于挂起状态。如果收到某不可靠信号，内核发现已经存在该信号处于未决状态，就会简单地丢弃该信号。因此发送不可靠信号，信号可能会丢失，

即内核递送给目标进程的次数，可能小于信号发送的次数。

对于可靠信号，内核内部有队列来维护，如果收到可靠信号，内核会将信号挂到相应的队列中，因此不会丢失。严格说来，内核也设有上限，挂起信号的个数也不能无限制地增大，因此只能说，在一定范围之内，可靠信号不会被丢弃。



注意 如果细心观察从kill-l列出的信号，可以看出，其中少了32号信号和33号信号。这两个信号（SIGCANCEL和SIGSETXID）被NPTL这个线程库征用了，用来实现线程的取消。从内核层来说，32号信号应该是最小的实时信号（SIGRTMIN），但是由于32号和33号被glibc内部征用了，所以glibc将SIGRTMIN设置成了34号信号。

6.5 传统信号的特点

前文提到过，signal是一个古老的机制，早期的信号在使用过程中，暴露出了一些弊端，那么早期的信号机制有什么弊端，表现出了什么样的行为模式呢？今天Linux下的glibc提供的信号函数是否解决了这些弊端，它又表现出了什么样的行为模式呢？下面来一探究竟。

传统的signal机制，分为System V风格和BSD风格的signal。

glibc提供了signal函数来注册用户定义的信号处理函数，代码如下：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

除此以外，Linux还提供了如下两个接口供我们“考古”，下面来探查一下signal机制的演化：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t sysv_signal(int signum, sighandler_t handler);
sighandler_t bsd_signal(int signum, sighandler_t handler);
```

从接口上看，存在4种signal函数，见表6-8。

表6-8 四种signal函数

函 数	说 明
syscall(SYS_signal,signo,func)	signal 系统调用
signal()	glibc 的 signal 函数
sysv_signal()	System V 风格的 signal 函数
bsd_signal()	BSD 风格的 signal 函数


接下来用实验的方法，测试各种不同的信号机制表现出来的行为模式，帮助大家体会传统信号的特点和弊端，以及学习Linux下glibc提供的signal函数的行为特性：

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/syscall.h>
#define MSG "OMG , I catch the signal SIGINT\n"
#define MSG_END "OK,finished process signal SIGINT\n"
int do_heavy_work()
{
    int i ;
    int k;
    srand(time(NULL));
    for(i = 0 ; i < 100000000;i++)
    {
        k = rand()%1234589;
    }
    return 0;
}
void signal_handler(int signo)
{
    write(2,MSG,strlen(MSG));
    do_heavy_work();
    write(2,MSG_END,strlen(MSG_END));
}
int main()
{
    char input[1024] = {0};
    #if defined SYSCALL_SIGNAL_API
    if(syscall(SYS_signal ,SIGINT,signal_handler) == -1)
    #elif defined SYSV_SIGNAL_API
    if(sysv_signal(SIGINT,signal_handler) == SIGERR)
    #elif defined BSD_SIGNAL_API
    if(bsd_signal(SIGINT,signal_handler) == SIGERR)
    #else
    if(signal(SIGINT,signal_handler) == SIG_ERR)
    #endif
    {
        fprintf(stderr,"signal failed\n");
        return -1;
    }
    printf("input a string:\n");
    if(fgets(input,sizeof(input),stdin)== NULL)
    {
        fprintf(stderr,"fgets failed(%s)\n",strerror(errno));
        return -2;
    }
    else
    {
        printf("you entered:%s",input);
    }
    return 0;
}
```

斜体的地方是这个测试程序的核心，这个函数分别采用了Linux操作系统提供的signal系统调用、System V风格的sysv_signal、BSD风格的bsd_signal，还有glibc提供的标准API signal函数。下面来分别体会它们之间的不同之处。

```
gcc -o syscall signal -DSYSCALL_SIGNAL_API signal_comp.c
gcc -o sysv_signal -DSYSV_SIGNAL_API signal_comp.c
gcc -o bsd_signal -DBSD_SIGNAL_API signal_comp.c
gcc -o glibc_signal signal_comp.c
```

这里分别生成了4种风格的测试程序，接下来就可以验证它们的特性了。

 **注意** 因为在x86_64位系统上，glibc的头文件并没有声明signal系统调用，因此，无法使用syscall函数来调用signal系统调用。详情可以参阅bits/syscall.h。不得已，只能在32位机器上做测试，比较四种函数语义上的差别。后面的输出都是在32位机器上的输出，希望不会给大家带来困扰。

6.5.1 信号的ONESHOT特性

传统的System V风格的signal，其注册的信号处理函数是一次性的，信号递送给目标进程之后，信号处理函数会变成默认值SIG_DFL。

```
manu@manu-hacks:~/code/c/self/signal$ ./sysv_signal
input a string:
hello
you entered:hello
manu@manu-hacks:~/code/c/self/signal$ ./sysv_signal
input a string:
hello^COMG , I catch the signal SIGINT
^C
manu@manu-hacks:~/code/c/self/signal$
```

可以看到第一次实验的时候，输入一个字符串，敲击回车，正常显示了输入的字符串。第二次输入结束之前，按Ctrl+C键，系统会向进程发送SIGINT信号，进程收到信号后，执行了信号处理函数（打印出了OMG, I catch the signal SIGINT），再次向进程发送SIGINT信号，进程就退出了。

可见，在System V风格的信号处理机制中，安装的信号处理函数是一次性的，内核把信号递送出去后，信号处理函数恢复成默认值SIG_DFL。因为SIGINT信号的默认处理是终止进程，所以进程就退出了。

Linux系统调用也是如此，信号处理函数同样是一次性的：

```
manu@manu-hacks:signal$ ./systemcall_signal
input a string:
hello
you entered:hello
manu@manu-hacks:signal$ ./systemcall_signal
input a string:
hello^COMG , I catch the signal SIGINT
^C
manu@manu-hacks:signal$
```

对于这种风格，内核中有个很形象的宏来描述这种行为模式，即SA_ONESHOT。

System V风格的singal处理机制就像图6-1中这种老式的单发手枪，每次射击完之后，都要重新上子弹，即信号处理函数触发之后，要想重复触发，必须再次安装信号处理函数。



图6-1 单发手枪，发射完毕，需要重新添加子弹

对于信号而言，是用标志位来控制信号的ONESHOT行为模式的，这个标志位是：

```
/*架构相关，对于
```

```
x86平台
```

```
*/
#define SA_RESETHAND    0x80000000u
#define SA_ONESHOT SA_RESETHAND
```

当内核递送信号给进程时，如果发现同时满足以下两个条件，则会将信号处理函数恢复成默认函数：

- 信号处理函数不是默认值。
- 信号处理函数的标志位中，SA_ONESHOT标志置位。

这部分控制逻辑，出现于内核的get_signal_to_deliver函数中：

```
int get_signal_to_deliver(signinfo_t *info, struct k_sigaction *return_ka,
                          struct pt_regs *regs, void *cookie)
{
    ...
    if (ka->sa.sa_handler == SIG_IGN) /* Do nothing. */
        continue;
    if (ka->sa.sa_handler != SIG_DFL) {
        /* Run the handler. */
        *return_ka = *ka;
        if (ka->sa.sa_flags & SA_ONESHOT)
            ka->sa.sa_handler = SIG_DFL;
        break; /* will return non-zero "signr" value */
    }
    ...
}
```

使用strace来追踪sysv_signal的执行，可以看到有如下的系统调用：

```
rt_sigaction(SIGINT, {0x8048756, [], SA_INTERRUPT|SA_NODEFER|SA_RESETHAND}, {SIG_DFL, [], 0}, 8) = 0
```

BSD风格的signal和glibc的signal函数已经不存在ONESHOT的问题了，代码如下所示：

```
manu@manu-hacks:signal$ ./bsd_signal
input a string:
hello^COMG , I catch the signal SIGINT
^COK,finished process signal SIGINT
OMG , I catch the signal SIGINT
OK,finished process signal SIGINT
^COMG , I catch the signal SIGINT
OK,finished process signal SIGINT
manu@manu-hacks:signal$ ./glibc_signal
input a string:
hello^COMG , I catch the signal SIGINT
^COK,finished process signal SIGINT
OMG , I catch the signal SIGINT
^COK,finished process signal SIGINT
OMG , I catch the signal SIGINT
OK,finished process signal SIGINT
```

通过strace追踪bsd_signal和glibc_signal执行的系统调用，可以看到，两者调用rt_sigaction系统调用时都没有设置SA_ONESHOT的标志位。

```
rt_sigaction(SIGINT, {0x8048736, [INT], SA_RESTART}, {SIG_DFL, [], 0}, 8) = 0
```



注意 通过strace追踪bsd_signal和glibc_signal可以看出，两者都调用了rt_sigaction系统调用，并且参数完全一致，表明在我的机器上，glibc的信号函数使用了BSD signal的语义。但是由于signal函数历史悠久，源远流长，在不同的平台上signal函数的语义可能并不相同。在相同的Linux平台上，由于glibc版本的差异，提供的signal函数的语义也有差异。在早期的libc4和libc5中，signal函数的语义是System V风格的。因此，从可移植的角度来看，不应该使用signal函数。

6.5.2 信号执行时屏蔽自身的特性

在执行信号处理函数期间，很有可能会收到其他的信号，当然也有可能再次收到正在处理的信号。如果在处理A信号期间再次收到A信号，会发生什么呢？

对于传统的System V信号机制，在信号处理期间，不会屏蔽对应的信号，而这就会引起信号处理函数的重入。这算是传统的System V信号机制的另一个弊端了。BSD信号处理机制修正了这个缺陷。当然了，BSD信号处理机制只是屏蔽了当前信号，并没有屏蔽当前信号以外的其他信号。

来比较下System V和BSD signal机制的区别。

System V风格的系统调用：

```
rt_sigaction(SIGINT, {0x8048756, [], SA_INTERRUPT|SA_NODEFER|SA_RESETHAND}, {SIG_DFL, [], 0}, 8) = 0
```

BSD风格的系统调用：

```
rt_sigaction(SIGINT, {0x8048736, [INT], SA_RESTART}, {SIG_DFL, [], 0}, 8) = 0
```

在上面的输出中，中括号内的是信号执行期间需要暂时屏蔽的信号。

BSD风格的信号处理机制，在安装信号的时候，会将自身这个信号添加到信号处理函数的屏蔽集合中。如果在执行A信号的信号处理函数期间，再次收到A信号，那么当前的A信号处理流程则不会被新来A信号打断。简单地说，就是不会嵌套了。

System V风格的信号，在其信号处理期间没有屏蔽任何信号，换句话说，执行信号处理函数期间，处理流程可以被任意信号中断，包括正在处理的信号。

从前面的实验可以看出，BSD风格的信号处理函数“OMG, I catch the signal SIGINT”，以及“OK, finished process signal SIGINT”总是成对出现的，不可能连续出现两个“OMG, I catch the signal SIGINT”，原因就是SIGINT信号在信号处理函数执行期间被暂时屏蔽了。

内核是如何做到这一点的？

完整的信号递送流程大致如此：内核首先调用`get_signal_to_deliver`，在挂起的信号集合中选择一个信号，递送给进程，选择完毕后，调用`handler_signal`函数。`handler_signal`函数的作用是为执行信号处理函数做准备。

```
void handler_signal(int sig, siginfo_t *info, struct k_sigaction *ka,
```

```
        struct pt_regs *regs, int stepping)
{
    sigset_t blocked;
    ...
    clear_restore_sigmask();
    sigorsets(&blocked, &current->blocked, &ka->sa.sa_mask);
    if (!(ka->sa.sa_flags & SA_NODEFER))
        sigaddset(&blocked, sig);
    set_current_blocked(&blocked);
    tracehook_signal_handler(sig, info, ka, regs, stepping);
}
```

从上面代码中不难看出，如果信号没有设置SA_NODEFER标志位，正在处理的信号就必须在信号处理程序执行期间被阻塞。

System V风格的signal机制为何会出现不屏蔽自身信号的情况？原因就是sysv_signal函数，在调用rt_sigaction系统调用时加上了SA_NODEFER标志位，如下：

```
rt_sigaction(SIGINT, {0x8048756, [], SA_INTERRUPT|SA_NODEFER

|SA_RESETHAND}, {SIG_DFL, [], 0}, 8) = 0
```

6.5.3 信号中断系统调用的重启特性

系统调用在执行期间，很可能会收到信号，此时进程可能不得不从系统调用中返回，去执行信号处理函数。对于执行时间比较久的系统调用（如wait、read等）被信号中断的可能性会大大增加。系统调用被中断后，一般会返回失败，并置错误码为EINTR。

如果程序员希望处理完信号之后，被中断的系统调用能够重启，则需要通过判断errno的值来解决，即如果发现错误码是EINTR，就重新调用系统调用。来看下面的例子：

```
manu@manu-hacks:~/code/c/self/signal$ ./sysv_signal
input a string:
^COMG , I catch the signal SIGINT
OK,finished process signal SIGINT
fgets failed(Interrupted system call)
```

通过strace可以看到，fgets调用了read系统调用，而read系统调用因为等待用户输入而陷入长时间的阻塞。在阻塞过程中，收到了一个SIGINT信号，导致read系统调用被中断，返回了错误码EINTR。

Linux世界中的很多系统调用都会遭遇这种情景，尤其是read、wait这种可能比较耗时的系统调用。《Unix系统编程：通信、并发和线程》一书中存在很多类似的例子：

```
pid_t r_wait(int *stat_loc)
{
    int retval;
    while(((retval = wait(stat_loc)) == -1 && (errno == EINTR)){
        ;
    }
    return retval;
}
```

这种封装就是用来应对系统调用被信号中断的场景的。当系统调用被信号中断时，程序并不认为这是一种无法处理的错误，相反，程序完全可以通过重新调用系统调用，来完成其想做的事情。

在System V信号机制下，系统调用如果被信号中断，则会返回-1，并置errno为EINTR，而不会主动重启被信号中断的系统调用。

细细想来，如果所有的系统调用都要判断返回值是否为EINTR，是的话，则重启系统调用，那么程序员就太累了。BSD风格的signal机制提供了另外一种思路，即如果系统调用被信号中断，内核会在信号处理函数结束之后，自动重启系统调用，无须程序员再次调用系统调用。

Linux操作系统提供了一个标志位SA_RESTART来告诉内核，被信号中断后是否要重启系统调用。如果该标志位为1，则表示如果系统调用被信号中断，那么内核会自动重启系统调用。

BSD风格的signal函数和glibc的函数，毫无意外地都带有该标志位：

```
rt_sigaction(SIGINT, {0x8048736, [INT], SA_RESTART}, {SIG_DFL, [], 0}, 8) = 0
```

由于BSD风格的signal存在这个标志SA_RESTART，因此fgets不会像System V的signal一样，返回错误码：

```
manu@manu-hacks:~/code/c/self/signal$ ./bsd_signal
input a string:
hello^COMG , I catch the signal SIGINT
OK,finished process signal SIGINT
^COMG , I catch the signal SIGINT
OK,finished process signal SIGINT
```

非常不幸的是，并不是所有的系统调用对信号中断都表现出同样的行为。某些系统调用哪怕设置了SA_RESTART的标志位，也绝不会自动重启。

那么问题就来了，在Linux下，如果信号处理函数设置了SA_RESTART，哪些阻塞型的系统调用遭到信号中断后，可以自动重启，哪些系统调用又是死活也无法自动重启的呢？

表6-9列出了设置SA_RESTART标志位后，可以自动重启的阻塞型系统调用。

表6-9 设置了SA_RESTART标志位，中断后可以自动重启的系统调用

read	write	readv	writew
ioctl	open	wait	waitpid
waitid	accept（没有设置超时）	connect（没有设置超时）	recv（没有设置超时）
recvfrom（没有设置超时）	recvmsg（没有设置超时）	send（没有设置超时）	sendto（没有设置超时）
sendmsg（没有设置超时）	flock	fcntl: F_SETLKW	mq_receive
mq_timedreceive	mq_send	mq_timedsend	futex: FUTEX_WAIT

表6-10是设置了SA_RESTART标志位，也不会重启的系统调用。

表6-10 设置了SA_RESTART标志位，中断后也无法自动重启的系统调用

poll	epoll	select	pselect
epoll_wait	epoll_pwait	msgrecv	msgsnd
semop	semtimedop	clock_nanosleep	nanosleep
usleep	accept（有设置超时）	connect（有设置超时）	recv（有设置超时）
recvfrom（有设置超时）	recvmsg（有设置超时）	send（有设置超时）	sendto（有设置超时）
sendmsg（有设置超时）			

太多了，记不住怎么办？man来帮忙。通过man 7 signal就可以获得这些信息。

通过前面三节的测试，可以得到表6-11中的结论。

表6-11 4种信号函数的不同表现

信 号 机 制	是否有 ONESHOT 特性	执行期间是否屏蔽自身信号	是否重启系统调用
System V	YES	NO	NO
BSD	NO	YES	YES
system call	YES	NO	NO
glibc signal	NO	YES	YES

手册明确表示bsd_signal没有ONESHOT特性，信号处理函数不会reset成默认值，无须重复安装信号处理函数；信号处理函数期间，自身信号会被屏蔽；系统调用被中断，会重启系统调用。这三个特性都是可以保证的，但是glibc下signal函数就不一定了，这要取决于操作系统，取决于glibc的版本。这是signal函数被人诟病的一个重要原因。简言之，就是其历史负担太重。

通过前面的讨论可以发现，Linux系统会通过一些标志位和屏蔽信号集来完成对某些特性的控制。

·SA_ONESHOT（或SA_RESERTHAND）：将信号处理函数恢复成默认值。

·SA_NODEFER（或SA_NOMASK）：显式地告诉内核，不要将当前处理信号值添加进阻塞信号集。

·SA_RESTART：将中断的系统调用重启，而不是返回错误码EINTR。

6.6 信号的可靠性

6.4节讲信号的分类时提到过，传统的信号存在信号丢失的问题，因此被称为不可靠信号。为了对传统的不可靠信号有更直观的认识，下面来做一个简单的实验，让事实来说话。我们可以疯狂地向某个进程发送信号，然后通过比较信号发送的次数和信号处理函数执行的次数来验证是否存在信号丢失的问题。

6.6.1 信号的可靠性实验

信号作为一种进程间的通信方式，通常会期望发射N次信号，那么目标进程就执行信号处理函数N次。对于传统信号而言，实际情况又如何呢？来看看下面的示例代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
static int sig_cnt[NSIG];
static volatile sig_atomic_t get_SIGINT = 0;
void handler(int signo)
{
    if(signo == SIGINT)
        get_SIGINT = 1;
    else
        sig_cnt[signo]++;
}
int main(int argc, char* argv[])
{
    int i = 0;
    sigset_t blockall_mask ;
    sigset_t empty_mask ;
    printf("%s:PID is %d\n", argv[0], getpid());
    for(i = 1; i < NSIG; i++)
    {
        if(i == SIGKILL || i == SIGSTOP ||
           i == 32 || i == 33)
            continue;
        if(signal(i, &handler) == SIG_ERR)
        {
            fprintf(stderr, "signal for signo(%d) failed (%s)\n",
                    i, strerror(errno));
        }
    }
    if(argc > 1)
    {
        int sleep_time = atoi(argv[1]);
        sigfillset(&blockall_mask);
        if(sigprocmask(SIG_SETMASK, &blockall_mask, NULL) == -1)
        {
            fprintf(stderr, "setprocmask to block all signal failed(%s)\n",
                    strerror(errno));
            return -2;
        }
        printf("I will sleep %d second\n", sleep_time);
        sleep(sleep_time);
        sigemptyset(&empty_mask);
        if(sigprocmask(SIG_SETMASK, &empty_mask, NULL) == -1)
        {
            fprintf(stderr, "setprocmask to release all signal failed(%s)\n",
                    strerror(errno));
            return -3;
        }
    }
    while(!get_SIGINT)
        continue ;
    printf("%-10s%-10s\n", "signo", "times");
    printf("-----\n");
    for(i = 1; i < NSIG ; i++)
    {
        if(sig_cnt[i] != 0 )
        {
            printf("%-10d%-10d\n", i, sig_cnt[i]);
        }
    }
    return 0;
}
```

下面来简单讲述这个程序。

如果执行时不带参数，那么进程会原地循环，直到收到SIGINT信号为止。在这期间，信号处理函数每执行一次，都会将收到信号的次数加1，进程结束前，会将各种信号收到的次数打印出来。

如果执行时带一个参数，那么这个参数的含义是屏蔽信号的时间N，首先将能够阻塞的信号全部阻塞，在信号阻塞期间，虽然会有进程向signal_receiver进程发送信号，但是内核并不会立即将收到的信号递送给进程。在沉睡N秒之后，解除阻塞，内核开始向signal_receiver进程递送信号。

再准备一个发送信号的程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
void usage()
{
    fprintf(stderr, "USAGE:\n");
    fprintf(stderr, "-----\n");
    fprintf(stderr, "signal_sender pid signo times\n");
}
int main(int argc, char* argv[])
{
    pid_t pid = -1;
    int signo = -1;
    int times = -1;
    int i;
    if (argc < 4)
    {
        usage();
        return -1;
    }
    pid = atoi(argv[1]);
    signo = atoi(argv[2]);
    times = atoi(argv[3]);
    if (pid <= 0 || times < 0 || signo < 1 ||
        signo >= 64 || signo == 32 || signo == 33)
    {
        usage();
        return -1;
    }
    printf("pid = %d, signo = %d, times = %d\n", pid, signo, times);
    for (i = 0; i < times; i++)
    {
        if (kill(pid, signo) == -1)
        {
            fprintf(stderr, "send signo(%d) to pid(%d) failed, reason(%s)\n",
                signo, pid, strerror(errno));
            return -2;
        }
    }
    fprintf(stdout, "done\n");
    return 0;
}
```

程序比较简单，接受三个参数：目标进程号、信号值和发送次数。有了这个工具，我们可以向目标进程signal_receiver连续发送任意次数的信号X。

首先，signal_receiver不带参数执行（即signal_receiver进程不会让信号阻塞一段时间），向signal_receiver连续发送信号，看看目标进程signal_receiver一共收到多少次信号：

终端

1:

```
manu@manu-hacks:signal$ ./signal_receiver
./signal_receiver:PID is 9937
```

向9937进程发送信号SIGUSR2 10000次，然后发送SIGINT信号1次来结束signal_receiver进程，下面查看signal_receiver进程一共收到多少次信号：

终端

```
2
manu@manu-hacks:signal$ ./signal_sender 9937 12 10000
pid = 9937, signo = 12, times = 10000
done
manu@manu-hacks:signal$ ./signal_sender 9937 2 1
pid = 9937, signo = 2, times = 1
```

signal_receiver进程打印结果为：

终端

```
1
signo      times
-----
12         2488
```

可以看到我们发送12号信号10000次，可是signal_receiver只收到2488次，这个2488也不是固定的，如果多执行几次，你会看到每次收到的信号次数均不相同，如下：

```
signo      times
-----
12         2352
signo      times
-----
12         2403
```

可以看到收到信号的次数是不一定的，但是都不等于发送信号的次数。再进一步，让信号接收进程屏蔽信号一段时间，在这段时间内，发送信号，查询信号处理函数被触发的次数：

终端

```
1
manu@manu-hacks:signal$ ./signal_receiver 30
./signal_receiver:PID is 27639
I will sleep 30 second终端

2
manu@manu-hacks:signal$ ./signal_sender 27639 10 10000
pid = 27639, signo = 10, times = 10000
done
manu@manu-hacks:signal$ ./signal_sender 27639 36 10000
pid = 27639, signo = 36, times = 10000
done终端
```

```
1
signo      times
-----
10         1
36         10000
```

从上面的例子可以看出，如果进程将信号屏蔽一段时间，在此期间向目标进程发送SIGUSR2信号10000次，在解除屏蔽之后，信号处理函数只触发了一次。

那么可靠信号的表现又如何呢？实验中发送实时信号36共计10000次，解除屏蔽后信号处理函数共触发了10000次，没有丢失信号，所有信号都被递送给进程去处理了。

6.6.2 信号可靠性差异的根源

从上面的实验可以看出可靠信号和不可靠信号存在着不小的差异。不可靠信号，不能可靠地被传递给进程处理，内核可能会丢弃部分信号。会不会丢弃，以及丢弃多少，取决于信号到来和信号递送给进程的时序。而可靠信号，基本不会丢失信号。

之所以存在这种差异，是因为重复的信号到来时，内核采取了不同的处理方式。从内核收到发给某进程的信号，到内核将信号递送给该进程，中间有个时间窗口。在这个时间窗口内，内核会负责记录收到的信号信息，这些信号被称为挂起信号或未决信号。但是对于可靠信号和不可靠信号，内核采取了不同的记录方式。

内核中负责记录挂起信号的数据结构为sigpending结构体，定义代码如下：

```
struct sigpending {
    struct list_head list;
    sigset_t signal;
};
#define _NSIG 64
#define _NSIG_BPW 64
#define _NSIG_WORDS (_NSIG / _NSIG_BPW)
typedef struct {
    unsigned long sig[_NSIG_WORDS];
} sigset_t;
```

在sigpending结构体中，sigset_t类型的成员变量signal本质上是一个位图，用一个比特来记录是否存在与该位置对应的信号处于未决的状态。根据位图可以有效地判断某信号是否已经存在未决信号。因为共有64种不同的信号，因此对于64位的操作系统，一个无符号的长整型就足以描述所有信号的挂起情况了。

在sigpending结构体中，第一个成员变量是个链表头。内核定义了结构体sigqueue，代码如下：

```
struct sigqueue {
    struct list_head list;
    int flags;
    siginfo_t info;
    struct _user_struct *user;
};
```

该结构体中info成员变量详细记录了信号的信息。如果内核收到发给某进程的信号，则会分配一个sigqueue结构体，并将该结构体挂入sigpending中第一个成员变量list为表头的链表之中。

综上所述，内核的进程描述符提供了两套机制来记录挂起信号：位图和队列。可能有读者会问，存在两套机制，尤其是存在队列，不应该丢失信号啊！正常来讲，来一个信号，只须将信号的相关信息挂入队列之中，就可以确保信号不丢。的确如此，但是实际上，可靠信号和不可靠信号的处理方式不同，不可靠信号并没有充分的队列来确保信号不丢。

内核收到不可靠信号时，会检查位图中对应位置是否已经是1，如果不是1，则表示尚无该信号处于挂起状态，然后会分配sigqueue结构体，并将信号挂入链表之中，同时将位图对应位置置1。但是如果位图显示已经存在该不可靠信号，那么内核会直接丢弃本次收到的信号。换句话说，内核的sigpending链表之中，最多只会存在一个不可靠信号的sigqueue结构体。

内核收到可靠信号时，不论是否已经存在该信号处于挂起状态，都会为该信号分配一个sigqueue结构体，并将sigqueue结构体挂入sigpending的链表之中，以确保不会丢失信号。

那么可靠信号是不是可以无限制地挂入队列呢？也不是。实际上内核也做了限制，一个进程默认挂起信号的个数是有限的，超过限制，可靠信号也会变得没那么可靠了，也会丢失信号。让我们看看内核代码：

```
static struct sigqueue *
__sigqueue_alloc(int sig, struct task_struct *t, gfp_t flags, int override_rlimit)
{
    struct sigqueue *q = NULL;
    struct user_struct *user;
    .....

    rcu_read_lock();
    user = get_uid(__task_cred(t)->user);
    atomic_inc(&user->sigpending);
    rcu_read_unlock();
    if (override_rlimit ||
        atomic_read(&user->sigpending) <=

        task_rlimit(t, RLIMIT_SIGPENDING)

    ) {
        q = kmem_cache_alloc(sigqueue_cachep, flags);
    } else {
        print_dropped_signal(sig);
    }
    if (unlikely(q == NULL)) {
        atomic_dec(&user->sigpending);
        free_uid(user);
    } else {
        INIT_LIST_HEAD(&q->list);
        q->flags = 0;
        q->user = user;
    }
    return q;
}
```

加粗部分的逻辑，决定了实时信号也不能被无限制地挂起。该限制属于资源限制的范畴，该限制项（RLIMIT_SIGPENDING）限制了目标进程所属的真实用户ID信号队列中挂起信号的总数。

可以通过如下命令来查看系统的限制：

```
manu@manu-hacks:~$ ulimit -
```

```
a
...
pending signals          (-i) 15144
...
```

用上面的测试程序测试一下，看看实时信号是否也会丢失信号：

终端

```
1
manu@manu-hacks:signal$ ./signal_receiver 30
./signal_receiver:PID is 14699
I will sleep 30 second终端

2
manu@manu-hacks:signal$ ./signal_sender 14699 36 20000
pid = 14699,signo = 36,times = 20000
done
manu@manu-hacks:signal$ ./signal_sender 14699 2 1
pid = 14699,signo = 2,times = 1
done
manu@manu-hacks:signal$终端
```

```
1
signo      times
-----
36         15144
```

和预期的一样，向目标进程发送了实时信号36共计20000次，但目标进程只收到了15144次，超出限制的部分都被丢弃掉了。

这个挂起信号的上限值是可以修改的，可以用`ulimit-i unlimited`这个命令将进程挂起信号的最大值设为无穷大，从而确保内核不会主动丢弃实时信号。

6.7 信号的安装

前面讲了传统信号的很多弊端，讲了signal的兼容性问题，有问题就会有解决方案。对此，Linux提供了新的信号安装方法：**sigaction**函数。和**signal**函数相比，这个函数的优点在于语义明确，可以提供更精确的控制。

先来看一下**sigaction**函数的定义：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

上面给出的**sigaction**结构体的定义并非严格意义上的定义，即结构体必须要有上述的成员变量，但成员变量的具体顺序取决于实现。

顾名思义，**sa_mask**就是信号处理函数执行期间的屏蔽信号集。前文介绍**bsd_signal**的时候曾提到，为**SIGINT**安装处理函数时，内核会自动将**SIGINT**添加入屏蔽信号集，在**SIGINT**信号处理函数执行期间，**SIGINT**信号不会被递送给进程。但是，也仅仅是**SIGINT**，如果执行**SIGINT**信号处理函数期间，需要屏蔽**SIGHUP**、**SIGUSR1**等其他信号，那**bsd_signal**函数就爱莫能助了。这个屏蔽其他信号的需求对**sigaction**函数而言，根本就不是问题，只需如下代码即可做到：

```
struct sigaction sa;

sa.sa_mask = SIGHUP|SIGUSR1|SIGINT;
```

需要特别指出的是，并不是所有的信号都能被屏蔽。对于**SIGKILL**和**SIGSTOP**，不可以为它们安装信号处理函数，也不能屏蔽掉这些信号。原因是，系统总要控制某些进程，如果进程可以自行设计所有信号的处理函数，那么操作系统可能无法控制这些进程。换言之，操作系统是终极boss，需要杀死某些进程的时候，要能够做到，**SIGKILL**和**SIGSTOP**不能被屏蔽，就是为了防止出现进程无法无天而操作系统徒叹奈何的困境。



注意 **SIGKILL**和**SIGSTOP**也不是万能的。如果进程处于**TASK_UNINTERRUPTIBLE**的状态，进程就不会处理信号。如果进程失控，长期处于该状态，**SIGKILL**也无法杀死该进程。详情可以回顾第5章。

若通过sigaction强行给SIGKILL或SIGSTOP注册信号处理函数，则会返回-1，并置errno为EINVAL。

在sigaction函数接口中，比较有意思的是sa_flags。sigaction函数之所以可以提供更精确的控制，大部分都是该参数的功劳。下面简要介绍一下sa_flags的含义，其中很多标志位并不是新面孔，前面已经讨论过了。

（1）SA_NOCLDSTOP

这个标志位只用于SIGCHLD信号。4.7节“等待子进程”中曾经提到过，父进程可以监测子进程的三种事件：

- 子进程终止（即子进程死亡）
- 子进程停止（即子进程暂停）
- 子进程恢复（即子进程从暂停中恢复执行）

其中SA_NOCLDSTOP标志位是用来控制第二种和第三种事件的。即一旦父进程为SIGCHLD信号设置了这个标志位，那么子进程停止和子进程恢复这两件事情，就无须向父进程发送SIGCHLD信号了。

（2）SA_NOCLDWAIT

这个标志只用于SIGCHLD信号，它可控制上面提到的子进程终止时的行为。如果父进程为SIGCHLD设置了SA_NOCLDWAIT标志位，那么子进程退出时，就不会进入僵尸状态，而是直接自行了断。但是子进程还会不会向父进程发送SIGCHLD信号呢？这取决于具体的实现。对于Linux而言，仍然会发送SIGCHLD信号。这点和上面的SA_NOCLDSTOP略有不同。

（3）SA_ONESHOT和SA_RESETHAND

这两个标志位的本质是一样的，表示信号处理函数是一次性的，信号递送出去之后，信号处理函数便恢复成默认值SIG_DFL。

（4）SA_NODEFER和SA_NOMASK

这两个标志位的作用是一样的，在信号处理函数执行期间，不阻塞当前信号。

（5）SA_RESTART

这个标志位表示，如果系统调用被信号中断，则不返回错误，而是自动重启系统调用。

(6) SA_SIGINFO

这个标志位表示信号发送者会提供额外的信息。这种情况下，信号处理函数应该为三参数的函数，代码如下：

```
void handle(int, siginfo_t *, void *);
```

此处重点讲述一下带SA_SIGINFO标志位的信号安装方式。本章引言中提到过，signal的本质是一种进程间的通信。一个进程向另外一个进程发送信号，能够传递的信息，不仅仅是signo，它还可以发送更多的信息，而接收进程也能获取到发送进程的PID、UID及发送的额外信息。

来看下面的例子：

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
void sig_handler(int signo,siginfo_t *info,void *context)
{
    printf("\nget signal:%d\n",signo);
    printf("signal number is %d\n",info->si_signo);
    printf("pid=%d\n",info->si_pid);
    printf("sigval = %d\n",info->si_value.sival_int);
}
int main(void)
{
    struct sigaction new_action;
    sigemptyset(&new_action.sa_mask);
    new_action.sa_sigaction = sig_handler;
    new_action.sa_flags |= SA_SIGINFO|SA_RESTART;
    if(sigaction(36,&new_action,NULL)==-1){
        printf("set signal process mode\n");
        exit(1);
    }
    while(1)
        pause();
    printf("Done\n");
    exit(0);
}
```

这个例子比较简单，为36号信号注册了信号处理函数。因为sa_flags带上了SA_SIGINFO标志位，所以必须使用三参数的信号处理函数。

```
void sig_handler(int signo,siginfo_t *info,void *context)
```

本例中的信号处理函数中，info->si_pid记录着信号发送者的PID，info->si_value.sival_int是信号发送进程时额外发送的int值。发送进程和接收进程约定好，发送者使用sigqueue发送信号，同时带上int型的额外信息，接收进程就能获得发送进程的PID及int型的额外信息。

如果调用sigaction函数时，sa_flags带了SIGINFO标志位，那么进程可以获得哪些信息？6.8.3小节介绍sigqueue函数时，会展开讲述。

6.8 信号的发送

6.8.1 kill、tkill和tgkill

kill函数的接口定义如下：

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

注意，不能望文生义，将kill函数的作用理解为杀死进程。kill函数的作用是发送信号。kill函数不仅可以向特定进程发送信号，也可以向特定进程组发送信号。第一个参数pid的值，决定了kill函数的不同含义，具体来讲，可以分成以下几种情况。

- pid>0：发送信号给进程ID等于pid的进程。
- pid=0：发送信号给调用进程所在的同一个进程组的每一个进程。
- pid=-1：有权限向调用进程发送信号的所有进程发出信号，init进程和进程自身除外。
- pid<-1：向进程组-pid发送信号。

当函数成功时，返回0，失败时，返回-1，并置errno。常见的出错情况见表6-12。

表6-12 kill函数的错误码及说明

errno	说 明
EINVAL	无效的信号值
EPERM	该进程没有权限发送信号给目标进程
ESRCH	目标进程或进程组不存在

有一种情况很有意思，即调用kill函数时，第二个参数signo的值为0。众所周知，没有一个信号的值是为0的，这种情况下，kill函数其实并不是真的向目标进程或进程组发送信号，而是用来检测目标进程或进程组是否存在。如果kill函数返回-1且errno为ESRCH，则可以断定我们关注的进程或进程组并不存在。

发送信号的典型方法如下：

```
if(kill(3423,SIGUSR1) == -1)
{
    /*error handler*/
}
```

如何向线程发送信号？

Linux提供了tkill和tgkill两个系统调用来向某个线程发送信号：

```
int tkill(int tid, int sig);
int tgkill(int tgid, int tid, int sig);
```

这两个都是内核提供的系统调用，glibc并没有提供对这两个系统调用的封装，所以如果想使用这两个函数，需要采用syscall的方式，如下：

```
ret = syscall(SYS_tkill,tid,sig)
ret = syscall(SYS_tgkill,tgid,tid,sig)
```

等一下，为什么有了tkill，还要引入tgkill？

实际上，tkill是一个过时的接口，并不推荐使用它来向线程发送信号。相比之下，tgkill接口更加安全。tgkill系统调用的第一个参数tgid，为线程组中主线程的线程ID，或者称为进程号。这个参数表面看起来是多余的，其实它能起到保护的作用，防止向错误的线程发送信号。进程ID

或线程ID这种资源是由内核负责管理的，进程（或线程）有自己的生命周期，比如向线程ID为1234的线程发送信号时，很可能线程1234早就退出了，而线程ID 1234恰好被内核分配给了另一个不相干的进程。这种情况下，如果直接调用`kill`，就会将信号发送到不相干的进程上。为了防止出现这种情况，于是内核引入了`tgkill`系统调用，含义是向线程组ID是`tgid`、线程ID为`tid`的线程发送信号。这样，出现误杀的可能就几乎不存在了。

这两个函数都是Linux特有的，存在可移植性的问题。

6.8.2 raise函数

Linux提供了向进程自身发送信号的接口：**raise**函数，其定义如下：

```
#include <signal.h>
int raise(int sig);
```

这个接口对于单线程的程序而言，就相当于执行如下语句：

```
kill(getpid(),sig)
```

这个接口对于多线程的程序而言，就相当于执行如下语句：

```
pthread_kill(pthread_self(),sig)
```

执行成功的时候，返回0，否则返回非零的值，并置**errno**。如果**sig**的值是无效的，**raise**函数就将**errno**置为EINVAL。

值得注意的是，信号处理函数执行完毕之后，**raise**才能返回。

6.8.3 sigqueue函数

在信号发送的方式当中，sigqueue算是后起之秀，传统的信号多用signal/kill这两个函数搭配，完成信号处理函数的安装和信号的发送。后来因为signal函数的表达力有限，控制不够精准，所以引入了sigaction函数来负责信号的安装，与其对应的是，引入了sigqueue函数来完成实时信号的发送。当然了，sigqueue函数也能发送非实时信号。

sigqueue函数的接口定义如下：

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

sigqueue函数拥有和kill函数类似的语义，也可以发送空信号（信号0）来检查进程是否存在。和kill函数不同的地方在于，它不能通过将pid指定为负值而向整个进程组发送信号。

比较有意思的是函数的第三个入参，它指定了信号的伴随数据（或者称为有效载荷，payload），该参数的数据类型是联合体，定义代码如下：

```
union sigval {
    int sival_int;
    void *sival_ptr;
};
```

通过指定sigqueue函数的第三个参数，可以传递一个int值或指针给目标进程。考虑到不同的进程有各自独立的地址空间，传递指针到另一个进程几乎没有任何意义。因此sigqueue函数很少传递指针（sival_ptr），大多是传递整型（sival_int）。



注意 尽管跨进程使用signal中的指针sival_ptr没有任何意义，但sival_ptr字段并非百无一用。该字段可用于使用signal联合体的其他函数中，如POSIX计时器的timer_create函数和POSIX消息队列中的mq_notify函数。

sigval联合体的存在，扩展了信号的通信能力。一些简单的消息传递完全可以使用sigqueue函数来进行。比如，通信双方事先定义某些事件为不同的int值，通过sigval联合体，将事件发送给目标进程。目标进程根据联合体中的int值来区分不同的事件，做出不同的响应。但是这种方法传递的消息内容受到了限制，不容易扩展，所以不宜作为常规的通信手段。

下面的例子会使用sigqueue函数向目标进程发送信号，其中目标进程、信号值和发送次数都可指定，发送信号的同时，也发送了伴随数据。

```
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
void usage()
{
    fprintf(stderr, "sigqueue_send sig pid [times]\n");
}
int main(int argc, char* argv[])
{
    pid_t pid;
    int sig;
    int times = 0;
    union sigval mysigval;
    if (argc < 3)
    {
        usage();
        return -1;
    }
    pid = atoi(argv[1]);
    sig = atoi(argv[2]);
    if (argc >= 4)
    {
        times = atoi(argv[3]);
    }
    mysigval.sival_int = 123;
    if (sig < 0 || sig > 64 || times < 0)
    {
        usage();
        return -2;
    }
    int i = 0;
    for (i = 0; i < times; i++)
    {
        if (sigqueue(pid, sig, mysigval) != 0)
        {
            fprintf(stderr, "sigqueue failed (%s)\n", strerror(errno));
            return -3;
        }
    }
    return 0;
}
```

一般来说，sigqueue函数的黄金搭档是sigaction函数。在使用sigaction函数时，只要给成员变量sa_flags置上SA_SIGINFO的标志位，就可以使

用三参数的信号处理函数来处理实时信号。

```
struct sigaction act;

...

act.sa_flags |= SA_SIGINFO;
```

三参数的信号处理函数如下：

```
void handle(int, siginfo_t *info, void *ucontext);
```

siginfo_t结构体存在以下成员：

```
siginfo_t {
    int      si_signo;
    int      si_errno;
    int      si_code;
    int      si_trapno;
    pid_t    si_pid;
    uid_t    si_uid;
    union {
        sigval_t si_value;
        void *si_addr;
        ...
    }
}
```

这个结构体包含很多信息，目标进程可以通过该数据结构获取到如下的信息：

- si_signo: 信号的值。
- si_code: 信号来源，可以通过这个值来判断信号的来源，具体见表6-13。

表6-13 si_code的值及其含义

si_code	信 号 来 源
SI_USER	调用 kill 或 raise 的用户进程
SI_TKILL	调用 tkill 或 tgkill 的用户进程
SI_QUEUE	调用 sigqueue 函数的用户进程
SI_MESGQ	消息到达 POSIX 消息队列
SI_KERNEL	内核产生的信号
SI_ASYNCIO	异步 I/O 操作完成
SI_TIMER	POSIX 定时器到期

除此之外，一些特殊的信号会产生一些独特的si_code，来表示信号产生的根源或来源。

例如，如果无效地址对齐引发SIGBUS信号，si_code就会被置为BUS_ADRALN等。想进一步了解详情，可以查看glibc的bits/siginfo.h头文件。

- si_value: sigqueue函数发送信号时所带的伴随数据。
- si_pid: 信号发送进程的进程ID。
- si_uid: 信号发送进程的真实用户ID。
- si_addr: 仅针对硬件产生的信号SIGBUS、SIGFPE、SIGILL和SIGSEGV设置该字段，该字段表示无效的内存地址（SIGBUS和SIGSEGV）或导致信号产生的程序的指令地址（SIGFPE和SIGILL）。

三参数信号处理函数的第三个参数是void*类型的，其实它是一个ucontext_t类型的变量。

```
typedef struct ucontext
{
    unsigned long int uc_flags;
    struct ucontext *uc_link;
```

```
    stack_t uc_stack;
    mcontext_t uc_mcontext;
    __sigset_t uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
} ucontext_t;
```

这个结构体提供了进程上下文的信息，用于描述进程执行信号处理函数之前进程所处的状态。通常情况下信号处理函数很少会用到这个变量，但是该变量也有很精妙的应用，如下面的例子。

对于C程序员而言，基本每个人都会遇到段错误。一般情况下，段错误出现的原因是程序访问了非法的内存地址。当段错误发生时，操作系统会发送一个SIGSEGV信号给进程，导致进程产生核心转储文件并且退出。如何才能让进程先捕捉SIGSEGV信号，打印出有用的方便定位问题的信息，然后再优雅地退出呢？可以通过给SIGSEGV注册信号处理函数来实现，代码如下所示：

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#ifdef USE_GNU
#define __USE_GNU
#endif
#include <execinfo.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ucontext.h>
#include <unistd.h>
typedef struct _sig_ucontext {
    unsigned long uc_flags;
    struct ucontext *uc_link;
    stack_t uc_stack;
    struct sigcontext uc_mcontext;
    sigset_t uc_sigmask;
} sig_ucontext_t;
void crit_err_hdlr(int sig_num, siginfo_t * info, void * ucontext)
{
    void * array[50];
    void * caller_address;
    char ** messages;
    int size, i;
    sig_ucontext_t * uc;
    uc = (sig_ucontext_t *)ucontext;
    caller_address = (void *) uc->uc_mcontext.rip;
    fprintf(stderr, "signal %d (%s), address is %p from %p\n",
        sig_num, strsignal(sig_num), info->si_addr,
        (void *)caller_address);
    size = backtrace(array, 50);
    array[1] = caller_address;
    messages = backtrace_symbols(array, size);
    /* 跳过第一个栈帧

*/
    for (i = 1; i < size && messages != NULL; ++i)
    {
        fprintf(stderr, "[bt]: (%d) %s\n", i, messages[i]);
    }
    free(messages);
    exit(EXIT_FAILURE);
}
int crash()
{
    char * p = NULL;
    *p = 0;
    return 0;
}
int foo4()
{
    crash();
    return 0;
}
int foo3()
{
    foo4();
    return 0;
}
int foo2()
{
    foo3();
    return 0;
}
int fool()
{
    foo2();
    return 0;
}
int main(int argc, char ** argv)
{
    struct sigaction sigact;
    sigact.sa_sigaction = crit_err_hdlr;
    sigact.sa_flags = SA_RESTART | SA_SIGINFO;
    if (sigaction(SIGSEGV, &sigact, (struct sigaction *)NULL) != 0)
    {
        fprintf(stderr, "error setting signal handler for %d (%s)\n",
            SIGSEGV, strsignal(SIGSEGV));
        exit(EXIT_FAILURE);
    }
    fool();
    exit(EXIT_SUCCESS);
}
```

上面的函数利用了第三个参数里面的ucontext->uc_mcontext.rip字段，获取到了收到信号前的EIP寄存器的值，根据该值，可以将堆栈信息打印出来，输出如下：

```
manu@manu-hacks:~/code/me/aple/chapter_05$ ./print_bt
signal 11 (Segmentation fault), address is (nil) from 0x40089d
[bt]: (1) ./print_bt() [0x40089d]
[bt]: (2) ./print_bt() [0x40089d]
[bt]: (3) ./print_bt() [0x4008b5]
[bt]: (4) ./print_bt() [0x4008ca]
```



```
[bt]: (5) ./print_bt() [0x4008df]  
[bt]: (6) ./print_bt() [0x4008f4]  
[bt]: (7) ./print_bt() [0x400984]  
[bt]: (8) /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf5) [0x7f6a8fa88ec5]  
[bt]: (9) ./print_bt() [0x400679]
```

缺点是没有打印出函数名，只打印了指令的地址。我们固然可以使用objdump得到汇编文件，根据地址查找到各自的函数名，但是手工干预太多，效率太低。如果在编译的时候，带上-rdynamic选项，就可打印出函数的地址了，代码如下所示：

```
root@manu-hacks:~/code/c/self/signal# gcc -o print_bt print_core.c -rdynamic  
manu@manu-hacks:~/code/me/apple/chapter_05$ ./print_bt  
signal 11 (Segmentation fault), address is (nil) from 0x400c0d  
[bt]: (1) ./print_bt(crash+0x10) [0x400c0d]  
[bt]: (2) ./print_bt(crash+0x10) [0x400c0d]  
[bt]: (3) ./print_bt(foo4+0xe) [0x400c25]  
[bt]: (4) ./print_bt(foo3+0xe) [0x400c3a]  
[bt]: (5) ./print_bt(foo2+0xe) [0x400c4f]  
[bt]: (6) ./print_bt(foo1+0xe) [0x400c64]  
[bt]: (7) ./print_bt(main+0x89) [0x400cf4]  
[bt]: (8) /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf5) [0x7efe7d126ec5]  
[bt]: (9) ./print_bt() [0x4009e9]
```

这样就可以很清楚地看到堆栈调用的关系，方便进一步定位问题。

6.9 信号与线程的关系

前面也曾简单提到过多线程，比如如何向多线程中的某个线程发送信号，本节就来重点讲述多线程与信号的关系。

提到线程与信号的关系，必须先介绍下POSIX标准，POSIX标准对多线程情况下的信号机制提出了一些要求：

- 信号处理函数必须在多线程进程的所有线程之间共享，但是每个线程要有自己的挂起信号集合和阻塞信号掩码。

- POSIX函数kill/sigqueue必须面向进程，而不是进程下的某个特定的线程。

- 每个发给多线程应用的信号仅递送给一个线程，这个线程是由内核从不会阻塞该信号的线程中随意选出来的。

- 如果发送一个致命信号到多线程，那么内核将杀死该应用的所有线程，而不仅仅是接收信号的那个线程。

这些就是POSIX标准提出的要求，Linux也要遵循这些要求，那它是怎么做到的呢？

6.9.1 线程之间共享信号处理函数

对于进程下的多个线程来说，信号处理函数是共享的。

在Linux内核实现中，同一个线程组里的所有线程都共享一个struct sighand结构体。该结构体中存在一个action数组，数组共64项，每一个成员都是k_sigaction结构体类型，一个k_sigaction结构体对应一个信号的信号处理函数。

相关数据结构定义如下（这与架构相关，这里给出的是x86_64位下的定义）：

```
struct sigaction { _sig_handler_t sa_handler;
                  unsigned long sa_flags;
                  __sigrestore_t sa_restorer;
                  sigset_t sa_mask;
};
struct k_sigaction {
    struct sigaction sa;
};
struct sighand_struct {
    atomic_t count;
    struct k_sigaction action[_NSIG];
    spinlock_t siglock;
    wait_queue_head_t signalfd_wqh;
};
struct task_struct {
    ...
    struct sighand_struct *sighand;
}
```

多线程的进程中，信号处理函数相关的数据结构如图6-2所示。

内核中k_sigaction结构体的定义和glibc中sigaction函数中用到的struct sigaction结构体的定义几乎是一样的。通过sigaction函数安装信号处理函数，最终会影响到进程描述符中的sighand指针指向的sighand_struct结构体对应位置上的action成员变量。

在创建线程时，最终会执行内核的do_fork函数，由do_fork函数走进copy_sighand来实现线程组内信号处理函数的共享。创建线程时，CLONE_SIGHAND标志位是置位的。创建线程组的主线程时，内核会分配sighand_struct结构体；创建线程组内的其他线程时，并不会另起炉灶，而是共享主线程的sighand_struct结构体，只须增加引用计数而已。

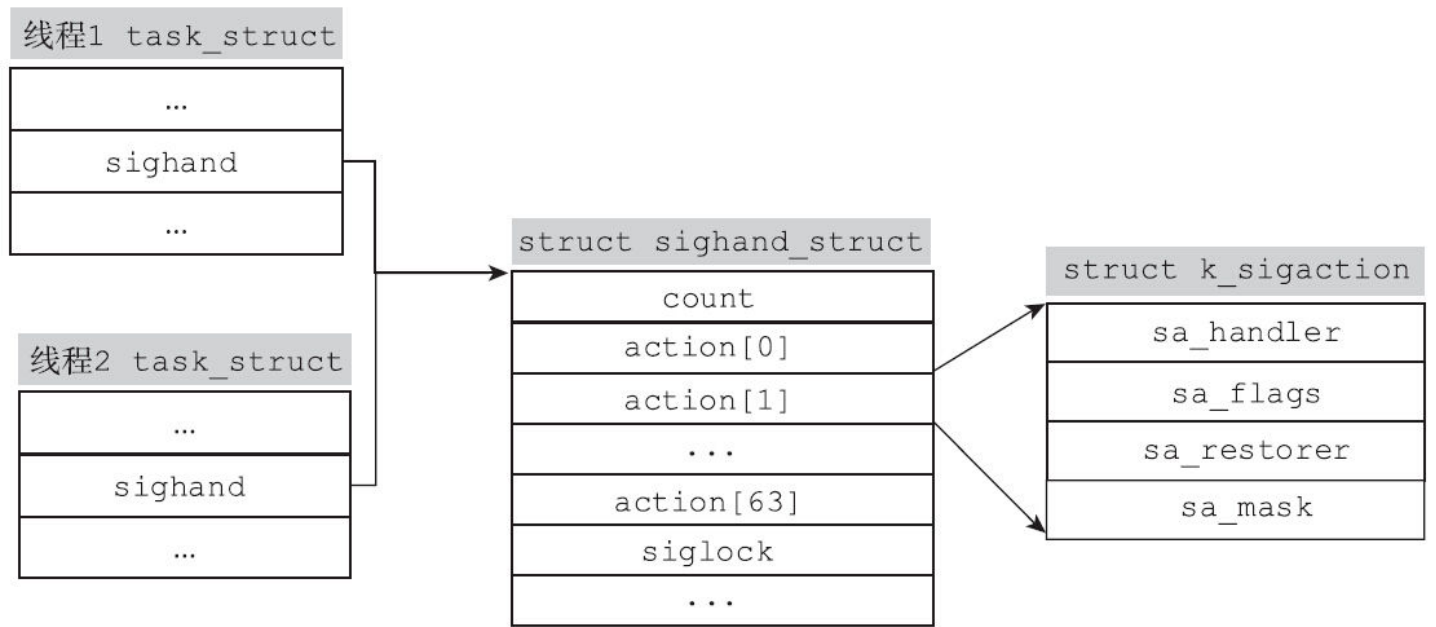


图6-2 同一进程里的多个线程共享信号处理函数

```
static int copy_sighand(unsigned long clone_flags,
                       struct task_struct *tsk)
{
    struct sighand_struct *sig;
    if (clone_flags & CLONE_SIGHAND) {
        //如果发现是线程，则直接将引用计数
```

++, 无须分配

sighand_struct结构

```
        atomic_inc(&current->sighand->count);
        return 0;
    }
    sig = kmem_cache_alloc(sighand_cache, GFP_KERNEL);
    rcu_assign_pointer(tsk->sighand, sig);
    if (!sig)
        return -ENOMEM;
    atomic_set(&sig->count, 1);
    memcpy(sig->action, current->sighand->action, sizeof(sig->action));
    return 0;
}
```

6.9.2 线程有独立的阻塞信号掩码

每个线程都拥有独立的阻塞信号掩码。在介绍这条性质之前，首先需要介绍什么是阻塞信号掩码。

就像我们开重要会议时要关闭手机一样，进程在执行某些重要操作时，不希望内核递送某些信号，阻塞信号掩码就是用来实现该功能的。如果进程将某信号添加进了阻塞信号掩码，纵然内核收到了该信号，甚至该信号在挂起队列中已经存在了相当长的时间，内核也不会将信号递送给进程，直到进程解除对该信号的阻塞为止。


开会时关闭手机是一种比较极端的例子。更合理的做法是暂时屏蔽部分人的电话。对于某些重要的电话，比如儿子老师的电话、父母的电话或老板的电话，是不希望被屏蔽的。信号也是如此。进程在执行某些操作的时候，可能只需要屏蔽一部分信号，而不是所有信号。

为了实现掩码的功能，Linux提供了一种新的数据结构：信号集。多个信号组成的集合被称为信号集，其数据类型为sigset_t。在Linux的实现中，sigset_t的类型是位掩码，每一个比特代表一个信号。

Linux提供了两个函数来初始化信号集，如下：

```
#include<signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

sigemptyset函数用来初始化一个空的未包含任何信号的信号集，而sigfillset函数则会初始化一个包含所有信号的信号集。



注意 必须要调用这两个初始化函数中的一个来初始化信号集，对于声明了sigset_t类型的变量，不能一厢情愿地假设它是空集合，也不能调用memset函数，或者用赋值为0的方式来初始化。

初始化信号之后，Linux提供了sigaddset函数向信号集中添加一个信号，同时还提供了sigdelset函数在信号集中移除一个信号：

```
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

为了判断某一个信号是否属于信号集，Linux提供了sigismember函数：

```
int sigismember(const sigset_t *set, int signum);
```

如果signum属于信号集，则返回1，否则返回0。出错的时候，返回-1。


有了信号集，就可以使用信号集来设置进程的阻塞信号掩码了。Linux提供了sigprocmask函数来做这件事情：

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

sigprocmask根据how的值，提供了三种用于改变进程的阻塞信号掩码的方式，见表6-14。

表6-14 sigprocmask函数中how的含义

how 参数	含 义
SIG_BLOCK	新的进程信号掩码是当前信号掩码与 set 指向信号集的并集，相当于在当前信号掩码中增加 set 中的信号
SIG_UNBLOCK	新的进程信号掩码是当前信号掩码与 set 指向信号集的补集的交集，相当于从当前信号掩码中删除 set 中的信号，解除对其的屏蔽
SIG_SETMASK	直接把进程的信号掩码设置成 set 指向的信号集



注意 我们知道SIGKILL信号和SIGSTOP信号不能阻塞，可是如果调用sigprocmask函数时，将SIGKILL信号和SIGSTOP信号添加进阻塞信号集中，会怎么样？

答案是不怎么样。sigprocmask函数不会报错，但是也不会将SIGKILL和SIGSTOP真的添加进阻塞信号集中。

对应的rt_sigprocmask系统调用会执行如下语句，剔除掉集合中的SIGKILL和SIGSTOP：

```
sigdelsetmask(&new_set, sigmask(SIGKILL)|sigmask(SIGSTOP));
```

对于多线程的进程而言，每一个线程都有自己的阻塞信号集：

```
struct task_struct{
    sigset_t blocked;
}
```

sigprocmask函数改变的是调用线程的阻塞信号掩码，而不是整个进程。sigprocmask出现得比较早，它出现在线程尚未引入Linux的时代。在单线程的时代，进程的阻塞信号掩码和线程的阻塞掩码是一回事，但是引入多线程之后，sigprocmask的语义就变成了设置调用线程的阻塞信号掩码。

为了更显式地设置线程的阻塞信号掩码，线程库提供了pthread_sigmask函数来设置线程的阻塞信号掩码：

```
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

事实上pthread_sigmask函数和sigprocmask函数的行为是一样的。

6.9.3 私有挂起信号和共享挂起信号

POSIX标准中有如下要求：对于多线程的进程，kill和sigqueue发送的信号必须面对所有的线程，而不是某个线程，内核是如何做到的呢？而系统调用tkill和tkill发送的信号，又必须递送给进程下某个特定的线程。内核又是如何做到的呢？

前面简单提到过内核维护有挂起队列，尚未递送进程的信号可以挂入挂起队列中。有意思的是，内核的进程描述符task_struct之中，维护了两套sigpending，代码如下所示：

```
struct task_struct{
    ...
    struct signal_struct *signal;
    struct sighand_struct *sighand;
    struct sigpending pending;    ...
}
struct signal_struct {...
    struct sigpending    shared_pending;
    ...
}
```

结构如图6-3所示。

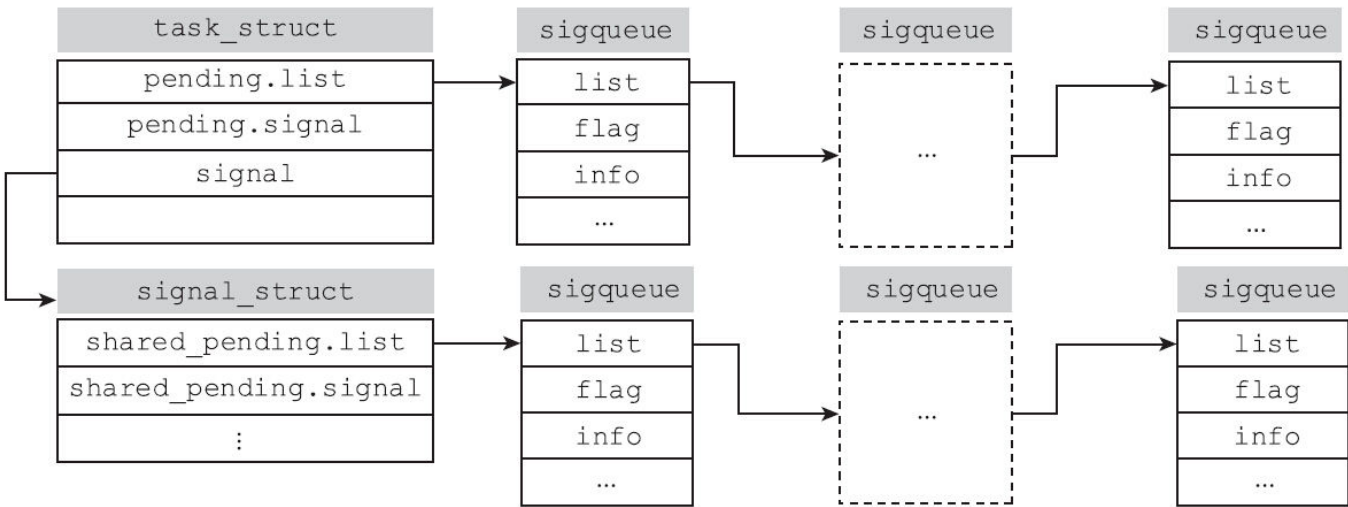


图6-3 信号挂起队列相关的数据结构

内核就是靠这两个挂起队列实现了POSIX标准的要求。在Linux实现中，线程作为独立的调度实体也有自己的进程描述符。Linux下既可以向进程发送信号，也可以向进程中的特定线程发送信号。因此进程描述符中需要有两套sigpending结构。其中task_struct结构体中的pending，记录的是发送给线程的未决信号；而通过signal指针指向signal_struct结构体的shared_pending，记录的是发送给进程的未决信号。每个线程都有自己的私有挂起队列（pending），但是进程里的所有线程都会共享一个公有的挂起队列（shared_pending）。

图6-4描述的是通过kill、sigqueue、tkill和tkill发送信号后，内核的相关处理流程。

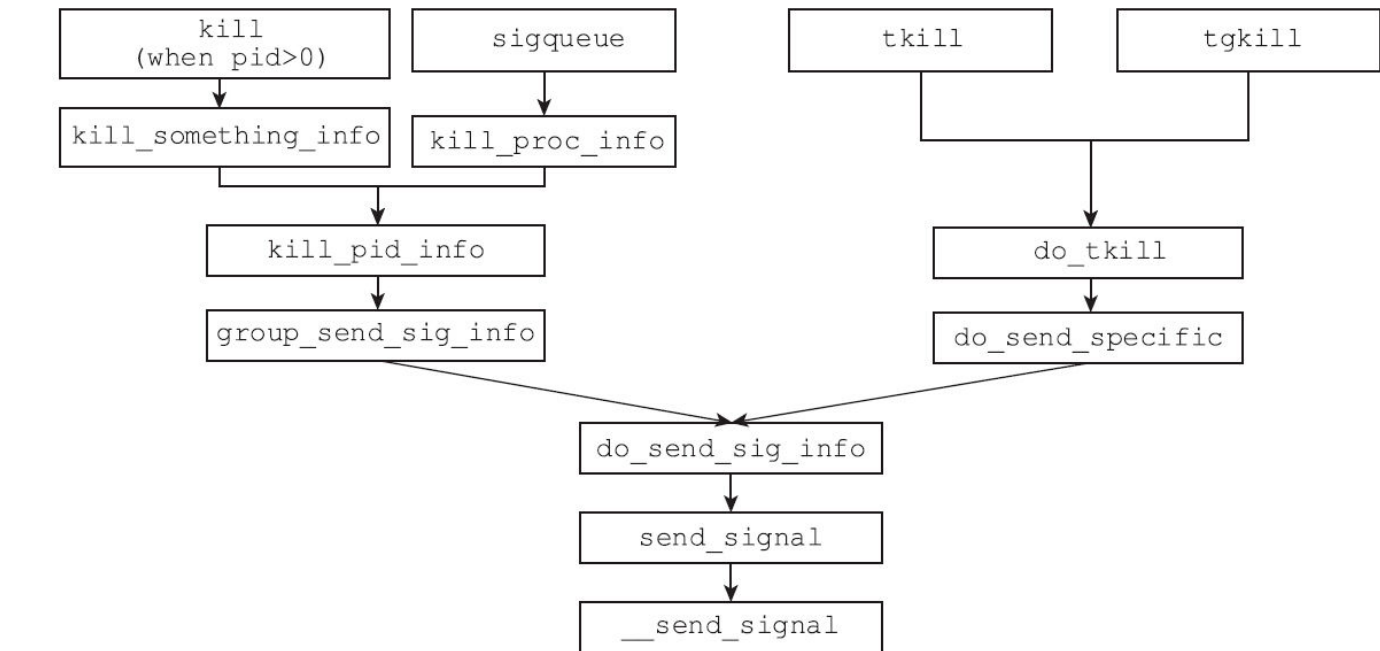


图6-4 发送信号相关的内核处理流程

从图6-4中可以看出，向进程发送信号也好，向线程发送信号也罢，最终都殊途同归，在do_send_sig_info函数处会师。尽管会师在一处，却还是存在不同。不同的地方在于，到底将信号放入哪个挂起队列。

在__send_signal函数中，通过group入参的值来判断需要将信号放入哪个挂起队列（如果需要进队列的话）。

```

static int __send_signal(int sig, struct siginfo *info, struct task_struct *t,
                        int group, int from_ancestor_ns)
{
    ...pending = group ? &t->signal->shared_pending : &t->pending;...
}
  
```

如果用户调用的是kill或sigqueue，那么group就是1；如果用户调用的是tkill或tgkill，那么group参数就是0。内核就是以此来区分该信号是发给进程的还是发给某个特定线程的，如表6-15所示。

表6-15 各种发送信号的函数与信号挂起队列的关系

函 数	group 的值	说 明
kill/sigqueue	1	需要的话，将信号挂入多个线程共享的 signal->shared_pending
tkill/tgkill	0	需要的话，将信号挂入线程私有的挂起队列 pending

上述情景并不难理解。多线程的进程就像是一个班级，进程下的每一个线程就像是班级的成员。kill和sigqueue函数发送的信号是给进程的，就像是优秀班集体的荣誉是颁发给整个班级的；tkill和tgkill发送的信号是给特定线程的，就像是三好学生的荣誉是颁发给学生个人的。

另一个需要解决的问题是，多线程情况下发送给进程的信号，到底由哪个线程来负责处理？这个问题就和高二（五）班荣获优秀班集体，由谁负责上台领奖一样。

内核是不是一定会将信号递送给进程的主线程？

答案是不一定。尽管如此，Linux还是采取了尽力而为的策略，尽量地尊重函数调用者的意愿，如果进程的主线程方便的话，则优先选择主线程来处理信号；如果主线程确实不方便，那就有可能由线程组里的其他线程来负责处理信号。

用户在调用kill/sigqueue函数之后，内核最终会走到__send_signal函数。在该函数的最后，由complete_signal函数负责寻找合适的线程来处理该信号。因为主线程的线程ID等于进程ID，所以该函数会优先查询进程的主线程是否方便处理信号。如果主线程不方便，则会遍历线程组中的其他线程。如果找到了方便处理信号的线程，就调用signal_wake_up函数，唤醒该线程去处理信号。

```

signal_wake_up(t, sig == SIGKILL);
  
```


如果线程组内全都不方便处理信号，complete函数也就当即返回了。

如何判断方便不方便？内核通过wants_signal函数来判断某个调度实体是否方便处理某信号：

```
static inline int wants_signal(int sig, struct task_struct *p)
{
    if (sigismember(&p->blocked, sig)) /*位于阻塞信号集，不方便

*/
    return 0;
    if (p->flags & PF_EXITING) /*正在退出，不方便

*/
    return 0;
    if (sig == SIGKILL) /*SIGKILL信号，必须处理

*/
    return 1;
    if (task_is_stopped_or_traced(p)) /*被调试或被暂停，不方便

*/
    return 0;
    return task_curr(p) || !signal_pending(p);
}
```

glibc提供了一个API来获取当前线程的阻塞挂起信号，如下：

```
#include <signal.h>
int sigpending(sigset_t *set);
```

该函数很容易产生误解，很多人认为该接口返回的是线程的挂起信号，即还没有来得及处理的信号，这种理解其实是错误的。

严格来讲，返回的信号集中的信号必须同时满足以下两个条件：

- 处于挂起状态。
- 信号属于线程的阻塞信号集。

看下内核的do_sigpending函数的内容就不难理解sigpending函数的含义了：

```
spin_lock_irq(&current->sighand->siglock);
sigorsets(&pending, &current->pending.signal,
          &current->signal->shared_pending.signal);
spin_unlock_irq(&current->sighand->siglock);
sigandsets(&pending, &current->blocked, &pending);
error = -EFAULT;
if (!copy_to_user(set, &pending, sigsetsize))
    error = 0;
```

因此，返回的挂起阻塞信号集合的计算方式是：

- 1) 进程共享的挂起信号和线程私有的挂起信号取并集，得到集合1。
- 2) 对集合1和线程的阻塞信号集取交集，以获得最终的结果。

从此处可以看出，sigprocmask函数会影响到sigpending函数的输出结果。

6.9.4 致命信号下，进程组全体退出

关于进程的退出，前面已经有所提及，Linux为了应对多线程，提供了`exit_group`系统调用，确保多个线程一起退出。对于线程收到致命信号的这种情况，操作是类似的。可以通过给每个调度实体的`pending`上挂上一个`SIGKILL`信号以确保每个线程都会退出。此处就不再赘述了。

6.10 等待信号

有时候，需要等待某种信号的发生。POSIX中的`pause`、`sigsuspend`和`sigwait`函数提供了三种方法，可以将进程暂时挂起，等待信号来临。

6.10.1 pause函数

pause函数将调用线程挂起，使进程进入可中断的睡眠状态，直到传递了一个信号为止。这个信号的动作或者是执行用户定义的信号处理函数，或者是终止进程。如果是执行用户自定义的信号处理函数，那么pause会在信号处理函数执行完毕后返回；如果是终止进程，pause函数就不返回了。如果内核发出的信号被忽略，那么进程就不会被唤醒。

pause函数的定义如下：

```
#include <unistd.h>
int pause (void);
```

比较有意思的是，pause函数如果可以返回，那它总是返回-1，并且errno为EINTR。

如果希望pause函数等待某个特定的信号，就必须确定哪个信号会让pause返回。事实上，pause并不能主动区分使pause返回的信号是不是正在等待的信号，我们必须间接地完成这个任务。

常用的方法是，在期待的特定信号的信号处理函数中，将某变量的值设置为1，待pause返回后，通过查看该变量的值是否为1来判定期待的特定信号是否被捕获，方法如下面的代码所示：

```
static volatile sig_atomic_t sig_received_flag = 0;
while(sig_received_flag == 0)
    pause();
```

如果只有等待的那个信号的处理函数会将sig_received_flag置成1，那么进程就会一直阻塞，直到接收到特定的信号为止。

看起来很美好，可是上面的逻辑是有漏洞的。检查sig_received_flag==0和调用pause之间存在一个时间窗口，如果在该时间窗口内收到信号，并且信号处理函数将sig_received_flag置1，那么主控制流根本就不知道这件事情，进程就会依然阻塞。也就是说，等待的信号已经到来，但是进程错过了。在收到下一个信号之前，pause函数不会返回，进程也就没有机会发现其实在等待的信号早就已经收到了。

因为检查和pause之间存在时间窗口，所以就有了错失信号的情况，如表6-16所示。

表6-16 错失等待信号的时序条件

时 间	sig_received_flag	程序控制流	信号处理函数
0	0	sig_received_flag == 0	
1	0		sig_received_flag = 1
2	1	pause	

下面通过另一个例子来描述一下pause的困境。程序执行过程中，关键部分不期望被信号打断，于是临时阻塞信号，关键部分完成之后，就解除信号的阻塞，然后暂停执行直到有信号到达为止：

```
/*关键代码结束

*/
sigprocmask(SIG_SETMASK, &orig_mask, NULL);
/*此处信号可能已经递送给进程了，导致

pause无法返回

*/
pause();
```

可以看到解除对特定信号的阻塞之后，调用pause之前，信号已经被递送给进程，这个信号已经错失了，pause无法等到这个信号，直到下一个信号递送给进程为止，pause函数都无法返回。这就违背了代码的本意：解除对信号的阻塞并且等待该信号的第一次出现。

要避免这种情况，必须将解除信号阻塞和挂起进程等待信号这两个动作封装成一个原子操作。这就是引入sigsuspend系统调用的原因。

6.10.2 sigsuspend函数

在pause之前传递信号是Linux早期遇到的一个困境，并没有好办法来解决这个问题。从本质上讲，必须将解除对信号的阻塞和挂起进程以等待信号的形式封装成一个原子操作，才能解决该问题，而sigsuspend函数就是为了解决这个难题而生的。

sigsuspend函数的定义如下：

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

如果信号终止了进程，那么sigsuspend函数不会返回。如果内核将信号递送给进程，并执行了信号处理函数，那么sigsuspend函数返回-1，并置errno为EINTR。如果mask指针指向的地址不是合法地址，那么sigsuspend函数返回-1，并置errno为EFAULT。

sigsuspend函数用mask指向的掩码来设置进程的阻塞掩码，并将进程挂起，直到进程捕捉到信号为止。一旦从信号处理函数中返回，sigsuspend函数就会把进程的阻塞掩码恢复为调用之前的老的阻塞掩码值。

简单地说，sigsuspend相当于以不可中断的方式执行下面的操作：

```
sigprocmask(SIG_SETMASK, &mask, &old_mask);
pause();
sigprocmask(SIG_SETMASK, &old_mask, NULL);
```

有了sigsuspend函数，就可以完成上一节pause完成不了的任务了。

```
static volatile sig_atomic_t sig_received_flag = 0;
sigset_t mask_all, mask_most, mask_old;
int signum = SIGUSR1;
sigfillset(&mask_all);
sigfillset(&mask_most);
sigdelset(&mask_most, signum);
sigprocmask(SIG_SETMASK, &mask_all, &mask_old);
/*不要忘记先判断，因为在
```

```
sigprocmask阻塞所有信号之前，
```

```
SIGUSR1可能已经被递送
```

```
*/
if(sig_received_flag == 0)
    sigsuspend(&mask_most);
sigprocmask(SIG_SETMASK, &mask_old, NULL);
```

假定等待特定信号SIGUSR1，首先要将所有的信号屏蔽掉，如果屏蔽信号之前，已经收到了

SIGUSR1，那么sig_received_flag会被设置为1，此时就不需要再调用sigsuspend了，我们已经等到了要等的信号SIGUSR1。如果没收到，则调用sigsuspend，将阻塞掩码设为mask_most，即将所有信号都屏蔽，只有SIGUSR1未被屏蔽。sigsuspend返回时，我们就可以确定，收到了信号SIGUSR1。此时，阻塞掩码也已经恢复成调用sigsuspend之前的mask_all了，然后显式地将阻塞掩码恢复成默认的阻塞掩码mask_old。

等一等，类似于上一节的代码，在判断之后、pause之前，有信号递送，会导致信号错失，那么在上面的代码中，判断sig_received_flag==1之后，调用sigsuspend函数之前，是否会有SIGUSR1被递送给进程，再次导致错失信号一次？答案是否定的，因为我们已经通过setprocmask函数阻塞了所有的信号，因此SIGUSR1没有机会被递送给进程。

上面的代码虽然完成了等待某特定信号的任务，但是它也有副作用，就是在等待特定信号期间，所有的其他信号都不能递送，原因是sigsuspend的mask阻塞了SIGUSR1以外的所有信号，导致其他信号无法正常递送。

下面的代码对这种情况做了改进：

```
static volatile sig_atomic_t sig_received_flag = 0;
sigset_t mask_blocked, mask_old, mask_unblocked;
int signum = SIGUSR1;
sigprocmask(SIGSETMASK, NULL, &mask_blocked);
sigprocmask(SIGSETMASK, NULL, &mask_unblocked);
sigaddset(&mask_blocked, signum);
sigdelset(&mask_unblocked, signum);
/*将
```

SIGUSR1添加到阻塞掩码中，确保下面判断

sig_received_flag和

sigsuspend之间不会收到

SIGUSR1信号，从而导致

SIGUSR1错失

```
*/
sigprocmask(SIG_BLOCK, &mask_blocked, &mask_old);
/*sigsuspend返回，可能是由其他信号引起的，
```

*因此需要再次判断

sig_received_flag是否置

```
1*/  
while(sig_received_flag == 0)  
    sigsuspend(&mask_unblocked);  
/*将信号恢复成默认值  
  
*/  
sigprocmask(SIG_SETMASK, &mask_old, NULL);
```

上面的例子不仅做到了等待特定信号SIGUSR1，而且期间如果有其他信号，也不会影响其他信号的递送。至此等待特定信号的任务算是圆满地解决了。

6.10.3 sigwait函数和sigwaitinfo函数

sigsuspend函数可以实现等待特定信号的任务，但是上面的示例过于繁复，不够直接。sigwait系列函数就可以比较优雅地等待某个特定信号的到来。sigwait系列函数提供了一种同步接收信号的机制，代码如下：

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);
```

这三个函数虽然接口上有差异，但是总体来说做的事情是一样的。信号集set里面的信号是进程关心的信号。当调用sigwait系列函数中的任何一个时，内核会查看进程的信号挂起队列（包括私有挂起队列和线程组共享的挂起队列），检查set中是否有信号处于挂起状态。如果有，那么sigwait相关的函数会立刻返回，并将信号从相应的挂起队列中移除；如果没有，进程就会陷入阻塞，进入可中断的睡眠状态，直到进程醒来，再次检查挂起队列。

上面是这三个函数的共同之处，不过它们在接口设计上有些许差异。

对于sigwait函数，成功返回时，返回值是0，并将导致函数返回的信号记录在sig指向的地址中。如果sigwait调用失败，则返回值不是-1，而是直接将errno返回。

sigwaitinfo函数是升级加强版的sigwait，通过它可以获取到信号相关的更多信息。当第二个siginfo_t结构体类型的指针info不是NULL时，内核会将信号相关的信息填入该指针指向的地址，从而获得导致函数返回的信号的信息。和sigwait函数不同，如果sigwaitinfo函数成功返回，那么返回值则是导致函数返回的信号的值（signo），而不是0；如果sigwaitinfo函数失败，则会返回-1，并置errno。

sigtimedwait函数和sigwaitinfo函数几乎是一样的，除了前者约定了一个timeout时间之外。如果到了timeout时间，还未等到set中的信号，sigtimedwait就不再继续等待了，而是返回-1，并置errno为EAGAIN。

sigwait系列函数的本质是同步等待信号的到达，所以不需要编写信号处理函数。需要提示的是，纵然某信号遭到了阻塞，sigwaitinfo依然可以获取等待信号。

看到这里，不知道读者有没有意识到，引入了sigwait系列函数之后，其实也引入了竞争。正常的信号处理流程，会从信号挂起队列中摘取信号递送给进程，而sigwait函数也会从信号挂起队列中摘取信号，返回给调用进程，两者成了抢生意的关系，如图6-5所示。

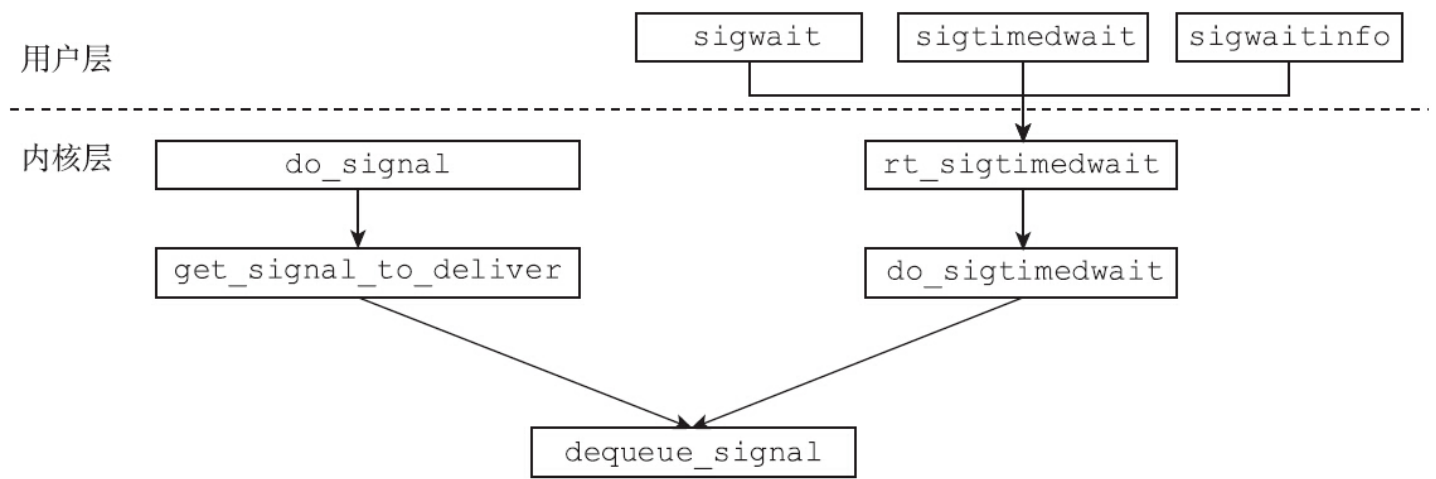


图6-5 sigwait和内核递送信号的竞争关系

所以在调用sigwait系列函数之前，首先需要将set中的信号阻塞，并将set中信号的独家经营权拿到手，否则，如果调用

sigwaitinfo之前或两次sigwaitinfo之间有信号到达，很有可能会被正常地递送给进程，进而执行注册的信号处理函数（如果有的话）或执行默认操作SIG_DFL。

sigwait系列函数的典型使用方式如下：

```
int sigusr1 = 0;
sigemptyset(&mask_sigusr1);
sigaddset(&mask_sigusr1, SIGUSR1);
sigprocmask(SIG_SETMASK, &mask_sigusr1, NULL);
while(1)
{
    sig = sigwaitinfo(&mask_sigusr1, &si);
    if(sig != -1)
    {
        sigusr1_count++;
    }
}
```

上面这个例子是统计收到SIGUSR1的次数。在调用sigwaitinfo之前，需要先调用sigprocmask将等待的信号屏蔽。

sigtimedwait函数的用法和sigwaitinfo函数类似，只不过timeout参数指定了最大的等待时长。如果调用超时却没有等到任何信号，那么sigtimedwait就返回-1，并且设errno为EAGAIN。

最后，SIGKILL和SIGSTOP不能等待，尝试等待SIGKILL和SIGSTOP会被忽略。原因和无法更改SIGKILL和SIGSTOP信号的信号处理函数一样。

6.11 通过文件描述符来获取信号

从内核2.6.22版本开始，Linux提供了另外一种机制来接收信号：通过文件描述符来获取信号即 `signalfd` 机制。

这个机制和 `sigwaitinfo` 非常地类似，都属于同步等待信号的范畴，都需要首先调用 `sigprocmask` 将关注的信号屏蔽，以防止被信号处理函数劫走。不同之处在于，文件描述符方法提供了文件系统的接口，可以通过 `select`、`poll` 和 `epoll` 来监控这些文件描述符。

`signalfd` 接口的定义如下：

```
#include <sys/signalfd.h>
int signalfd(int fd, const sigset_t *mask, int flags);
```

其中，`mask` 参数是信号集，表示关注信号的集合。这些信号的集合应该在调用 `signalfd` 函数之前，先调用 `sigprocmask` 函数阻塞这些信号，以防止被信号处理函数劫走。

首次创建时 `fd` 参数应该为 -1，该函数会创建一个文件描述符，用于读取 `mask` 中到来的信号。如果 `fd` 不是 -1，则表示是修改操作，一般是修改 `mask` 的值，此时 `fd` 是之前调用 `signalfd` 时返回的值。

第三个参数 `flags` 用来控制行为，目前支持的标志位如下。

· `SFD_CLOEXEC`：和普通文件的 `O_CLOEXEC` 一样，调用 `exec` 函数时，文件描述符会被关闭。

· `SFD_NONBLOCK`：控制将来的读取操作，如果执行 `read` 操作时，并没有信号到来，则立刻返回失败，并设置 `errno` 为 `EAGAIN`。

创建文件描述符后，可以使用 `read` 函数来读取到来的信号。提供的缓冲区大小一般要足以放下一个 `signalfd_siginfo` 结构体，该结构体一般包括如下成员变量：

```
struct signalfd_siginfo {
    uint32_t ssi_signo;
    int32_t ssi_errno;
    int32_t ssi_code;
    uint32_t ssi_pid;
    uint32_t ssi_uid;
    int32_t ssi_fd;
    uint32_t ssi_tid;
    uint32_t ssi_band;
    uint32_t ssi_overrun;
    uint32_t ssi_trapno;
    int32_t ssi_status;
    int32_t ssi_int;
    uint64_t ssi_ptr;
    uint64_t ssi_utime;
    uint64_t ssi_stime;
    uint64_t ssi_addr;
    uint8_t pad[X];
};
```

这个结构体和前面提到的`siginfo_t`结构体几乎可以一一对应。含义和`siginfo_t`中的成员也一样，在此就不再赘述了。

使用`signalfd`来接收信号的方法如下（此处忽略了一些异常处理）：

```
sigprocmask(SIG_BLOCK, &mask, NULL);
sfd = signalfd(-1, &mask, NULL);
for(;;)
{
    n = read(sfd, &fd_siginfo, sizeof(struct signalfd_siginfo));
    if(n != sizeof(struct signalfd_siginfo))
    {
        /*error handle*/
    }
    else{
        /*process the signal*/
    }
}
```

比较推荐的做法是用文件描述符`signalfd`和`sigwaitinfo`两种方法来处理信号，使用传统信号处理函数会因为异步带来很多问题，大量的函数因不是异步信号安全的，而无法用于信号处理函数。本节介绍的`signalfd`方法更加值得推荐，因为方法简单，且可以和`select`、`poll`和`epoll`函数配合使用，非常灵活。

6.12 信号递送的顺序

有一个非常有意思的话题，当有多个处于挂起状态的信号时，信号递送的顺序又是如何的呢？

信号实质上是一种软中断，中断有优先级，信号也有优先级。如果一个进程有多个未决信号，那么对于同一个未决的实时信号，内核将按照发送的顺序来递送信号。如果存在多个未决的实时信号，那么值（或者说编号）越小的越优先被递送。如果既存在不可靠信号，又存在可靠信号（实时信号），虽然POSIX对这一情况没有明确规定，但Linux系统和大多数遵循POSIX标准的操作系统一样，即将优先递送不可靠信号。

虽然是优先递送不可靠信号，但在不可靠信号中，不同信号的优先级又是如何的呢？内核如何实现这些这些优先级的顺序呢？

内核选择信号递送给进程的流程图如图6-6所示。



图6-6 内核选择信号的流程

下面来分析相关的代码：

```
int dequeue_signal(struct task_struct *tsk,
                  sigset_t *mask, siginfo_t *info)
{
    int signr;
    /* We only dequeue private signals from ourselves, we don't let
     * signalfd steal them
     */
    /*线程私有的挂起信号队列优先

*/
    signr = __dequeue_signal(&tsk->pending, mask, info);
    if (!signr) {
        signr = __dequeue_signal(&tsk->signal->shared_pending,
                                mask, info);
        ...
    }
}
```

前文讲过，线程的挂起信号队列有两个：线程私有的挂起队列（pending）和整个线程组共享的挂起队列（signal->shared_pending）。如上面的代码所示，选择信号的顺序是优先从私有的挂起队列中选择，如果没有找到，则从线程组共享的挂起队列中选择信号递送给线程。当然选择的时候需要考虑线程的阻塞掩码，属于阻塞掩码集中的信号不会被选出。

在挂起信号队列（无论是共享挂起队列还是私有挂起队列）中，选择信号的工作交给了next_signal函数，其逻辑如下：

```
int next_signal(struct sigpending *pending, sigset_t *mask)
{
    unsigned long i, *s, *m, x;
    int sig = 0;
    s = pending->signal.sig;
    m = mask->sig;
    /*
     * Handle the first word specially: it contains the
     * synchronous signals that need to be dequeued first.
     */
    x = *s &~ *m;
    if (x) {
        /*优先选择同步信号，所谓同步信号集合就是
        SIGSEGV.

        SIGBUS等六种信号

*/
        if (x & SYNCHRONOUS_MASK)

            x &= SYNCHRONOUS_MASK;
        /*小信号值优先递送的算法

*/
        sig = ffz(~x)

+ 1;
    return sig;
}
```

```

switch (_NSIG_WORDS) {
default:
    for (i = 1; i < _NSIG_WORDS; ++i) {
        x = *++s &~ *++m;
        if (!x)
            continue;
        sig = ffz(~x) + i*_NSIG_BPW + 1;
        break;
    }
    break;
case 2:
    x = s[1] &~ m[1];
    if (!x)
        break;
    sig = ffz(~x) + _NSIG_BPW + 1;
    break;
case 1:
    /* Nothing to do */
    break;
}
return sig;
}
#define SYNCHRONOUS_MASK \
(sigmask(SIGSEGV) | sigmask(SIGBUS) | sigmask(SIGILL) | \
 sigmask(SIGTRAP) | sigmask(SIGFPE) | sigmask(SIGSYS))

```

由于不同平台long的长度不同，所以算法略有不同，但是思想是一样的，如下。

- 1) 出现在阻塞掩码集中的信号不能被选出。
- 2) 优先选择同步信号，所谓同步信号指的是以下6种信号：

```
{SIGSEGV,SIGBUS,SIGILL,SIGTRAP,SIGFPE,SIGSYS}.
```

这6种信号都是与硬件相关的信号。

- 3) 如果没有上面6种信号，非实时信号优先；如果存在多种非实时信号，小信号值的信号优先。
- 4) 如果没有非实时信号，那么实时信号按照信号值递送，小信号值的信号优先递送。

通过下面的测试程序来验证是否如此：

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
static int sig_cnt[NSIG];
static number= 0 ;
int sigorder[128]={0};
#define MSG "%d:receiver signal %d\n"
void handler(int signo)
{
    /*此处最好判断一下

number的值，不要超出数组的长度

*/
    sigorder[number++] = signo;
}
int main(int argc,char* argv[])
{
    int i = 0;
    int k = 0;
    sigset_t blockall_mask ;
    sigset_t pending_mask ;
    sigset_t empty_mask ;
    struct sigaction sa ;
    sigfillset(&blockall_mask);
#ifdef USE_SIGACTION
    sa.sa_handler = handler;
    sa.sa_mask = blockall_mask ;
    sa.sa_flags = SA_RESTART ;
#endif
    printf("%s:PID is %d\n",argv[0],getpid());
    for(i = 1; i < NSIG; i++)
    {
        if(i == SIGKILL || i == SIGSTOP)
            continue;
#ifdef USE_SIGACTION
        if(sigaction(i,&sa, NULL)!=0)
#else
        if(signal(i,handler)== SIG_ERR)
#endif
        {
            fprintf(stderr,"sigaction for signo(%d) failed (%s)\n",i, strerror(errno));
            // return -1;
        }
    }
    int sleep_time = atoi(argv[1]);
    if(sigprocmask(SIG_SETMASK,&blockall_mask,NULL) == -1)
    {
        fprintf(stderr,"setprocmask to block all signal failed(%s)\n",strerror(errno));
        return -2;
    }
    printf("I will sleep %d second\n",sleep_time);
    sleep(sleep_time);

```

```
sigemptyset(&empty_mask);
if(sigprocmask(SIG_SETMASK, &empty_mask, NULL) == -1)
{
    fprintf(stderr, "setprocmask to release all signal failed(%s)\n", strerror(errno));
    return -3;
}
sleep(3)
for(i = 0 ; i< number ; i++)
{
    if(sigorder[i] != 0)
    {
        printf("#%d: signo=%d\n", i, sigorder[i]);
    }
}
return 0;
}
```

注意上面的代码必须要定义USE_SIGACTION宏，因为在执行信号处理函数期间，需要屏蔽掉其他信号，否则信号处理函数被其他信号打断，会导致无法得到信号的真实递送顺序。

上述程序首先会安装所有信号的信号处理函数（SIGKILL和SIGSTOP除外），然后阻塞所有信号，之后睡眠一段时间，在这段时间内，通过命令向进程发送各种信号，一旦睡眠结束，解除阻塞，信号就会被递送给进程，进程就会执行信号处理函数。信号处理函数是精心定制的，按照递送的顺序，被记录在静态数组中。只要按顺序打印出信号的值，就可获得信号的递送顺序：

```
gcc -o sigaction_delivery_order -DUSE_SIGACTION signal_delivery_order.c
```

向进程发送信号的脚本如下：

```
#!/bin/bash
./sigaction_delivery_order 30 &
signal_pid=$!
sleep 2
kill -10 $signal_pid
kill -3 $signal_pid
kill -12 $signal_pid
kill -11 $signal_pid
kill -39 $signal_pid
kill -2 $signal_pid
kill -5 $signal_pid
kill -4 $signal_pid
kill -36 $signal_pid
kill -24 $signal_pid
kill -38 $signal_pid
kill -37 $signal_pid
kill -31 $signal_pid
kill -8 $signal_pid
kill -7 $signal_pid
./tkill -p $signal_pid -s 44
```

tkill是发给具体线程的，信号会挂在线程私有的挂起信号队列上，所以会优先递送，因此44号信号应该第一个被递送；其他的信号中4=SIGILL、5=SIGTRAP、7=SIGBUS、8=SIGFPE、11=SIGSEGV、31=SIGSYS，这些都属于同步信号集合；紧随44号信号之后，按照从小到大的顺序递送；2、3、10、12、24作为非实时信号，再随其后被递送；最后是实时信号，按照从小到大的顺序（即36、37、38、39），依次递送给进程。

测试的输出结果如下所示：

```
root@manu-hacks:~/code/c/self/signal# ./test_order.sh
./sigaction_delivery_order:PID is 21897
sigaction for signo(32) failed (Invalid argument)
sigaction for signo(33) failed (Invalid argument)
I will sleep 30 second
root@manu-hacks:~/code/c/self/signal# #0: signo=44
#1: signo=4
#2: signo=5
#3: signo=7
#4: signo=8
#5: signo=11
#6: signo=31
#7: signo=2
#8: signo=3
#9: signo=10
#10: signo=12
#11: signo=24
#12: signo=36
#13: signo=37
#14: signo=38
#15: signo=39
```

和预想的一样，信号就是按照这四个优先级递送给进程的。

6.13 异步信号安全

设计信号处理函数是一件很头疼的事情，原因就藏在图6-7中。当内核递送信号给进程时，进程正在执行的指令序列就会被中断，转而执行信号处理函数。待信号处理函数执行完毕返回（如果可以返回的话），则继续执行被中断的正常指令序列。此时，问题就来了，同一个进程中出现了两条执行流，而两条执行流正是信号机制众多问题的根源。

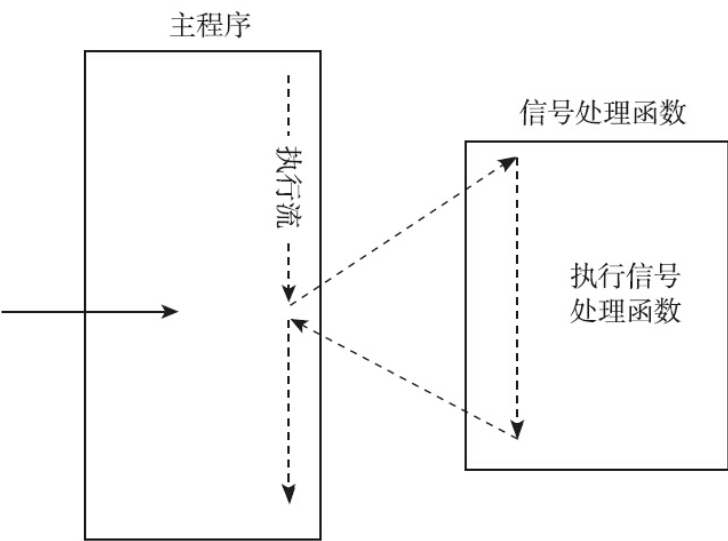


图6-7 进程收到信号的处理流程

在信号处理函数中有很多函数都不可以使用，原因就是它们并不是异步信号安全的，强行使用这些不安全的函数隐患重重，还可能带来很诡异的bug。

引入多线程后，很多库函数为了保证线程安全，不得不使用锁来保护临界区（见表6-17）。比如malloc就是一种典型的场景。

表6-17 锁来保证线程安全

时 间	线程 1 执行流	线程 2 执行流
0	Lock	Lock
1	临界区	
2	Unlock	
3		临界区
4		Unlock

加锁保护临界区的方法，虽然不可重入，却是实现线程安全的一种选择。但是这种方法无法保证异步信号安全见表6-18。

表6-18 锁无法保证异步信号安全

时 间	主程序执行流	信号处理函数执行流
0	Lock	
2	临界区	Lock ←死锁
3	Unlock	

还是以malloc为例，如果主程序执行流调用malloc已经持有了锁，但是尚未完成临界区的操作，这时候被信号中断，转而执行信号处理函数，如果信号处理函数中再次调用malloc加锁，就会发生死锁。

从上面的讨论可以看出，异步信号安全是一个很苛刻的条件。事实上只有非常有限的函数才能保证异步信号安全。

一般说来，不安全的函数大抵上可以分为以下几种情况：

- 使用了静态变量，典型的是`strtok`、`localtime`等函数。
- 使用了`malloc`或`free`函数。
- 标准I/O函数，如`printf`。

读者可以通过man 7 `signal`的`Async-signal-safe functions`小节查看异步信号安全的函数列表，在此就不罗列了。本书中有很多地方在信号处理函数中调用了`printf`函数，其实这是不对的，在真正的工程代码中，是不允许非异步信号安全的函数出现在信号处理函数中的。

在正常程序流里面工作得很正常的函数，在异步信号的条件下，会出现很诡异的bug。这种bug的触发，经常依赖信号到达的时间、进程调度等不可控制的时序条件，很难重现。因此编写信号处理函数就像将船驶入暗礁丛生的海域，不可不小心。

既然陷阱重重，那该如何使用信号机制呢？

1.轻量级信号处理函数

这是一种比较常见的做法，就是信号处理函数非常短，基本就是设置标志位，然后由主程序执行流根据标志位来获知信号已经到达。

这种做法可用伪代码的形式表示，如下：

```
volatile sig_atomic_t get_SIGINT = 0;
/*信号处理函数

*/
void sigint_handler(int sig)
{
    switch(sig){    case SIGINT:        get_SIGINT = 1        break;    ...}
}
/*主程序流是一个循环

*/
while(true)
{
    if (get_SIGINT==1)
    {
        /*在主程序流中处理

SIGINT*/
    }
    job = get_next_job();
    do_single_job(job);
}
```

这是一种常见的设计，信号处理函数非常简单，非常轻量，仅仅是设置了一个标志位。程序的主流程会周期性地检查标志，以此来判断是否收到某信号。若收到信号，则执行相应的操作，通常也会将标志重新清零。

一般来讲定义标志的时候，会将标志的类型定义成：

```
volatile sig_atomic_t flag;
```

`sig_atomic_t`是C语言标志定义的一种数据类型，该数据类型可以保证读写操作的原子性。而`volatile`关键字则是告诉编译器，`flag`的值是易变的，每次使用它的时候，都要到`flag`的内存地址去取。之所以这么做，是因为编译器会做优化，编译器如果发现两次取`flag`值之间，并没有代码修改过`flag`，就有可能将上一次的`flag`值拿来用。而由于主程序和信号处理不在一个控制流之中，因此编译器几乎总是会做这种优化，这就违背了设计的本意。因此使用`volatile`来保证主程序流能够看到信号处理函数对`flag`的修改。

2.化异步为同步

由于信号处理函数的存在，进程会同时存在两条执行流，这带来了很多问题，因此操作系统也想了一些办法，就是前面提到的`sigwait`和`sigwaitfd`机制。

`sigwait`的设计本意是同步地等待信号。在执行流中，执行`sigwait`函数会陷入阻塞，直到等待的信号降临。一般来讲，`sigwait`用在多线程的程序中，而等待信号降临的使命，一般落在主线程身上。具体做法如下：

```
sigfillset(&set_all);
sigprocmask(SIG_SETMASK, &set_all, NULL);
for(;;)
{
    ret = sigwait(&set_all, &signo);

    /*处理收到的

signo*/
}
```

sigwait虽然化异步为同步，但是也废掉了一条执行流。signalfd机制则提供了另外一种思路：

```
#include <sys/signalfd.h>
int signalfd(int fd, const sigset_t *mask, int flags);
```

具体步骤如下：

- 1) 将关心的信号放入集合。
- 2) 调用sigprocmask函数，阻塞关心的信号。
- 3) 调用signalfd函数，返回一个文件描述符。

有了文件描述符，就可以使用select/poll/epoll等I/O多路复用函数来监控它。这样，当信号来临时，就可以通过read接口来获取到信号的相关信息：

```
struct signalfd_info signalfd_info;
read(signal_fd, &signalfd_info, sizeof(struct signalfd_info));
```

在引入signalfd机制以前，有一种很有意思的化异步为同步的方式被广泛使用。这种技术被称为“self-pipe trick”。简单地讲，就是打开一个无名管道，在信号处理函数中向管道写入一个字节（write函数是异步信号安全的），而主程序从无名管道中读取一个字节。通过这种方式也做到了在主程序流中处理信号的目的。

《Linux高性能服务器编程》一书中，在“统一事件源”一节中详细介绍了这个技术。不过使用的不是无名管道，而是socketpair函数。

```
static int pipefd[2]
/*信号处理函数中，向

socketpair中写入

1个字节，即信号值

*/
void sig_handler(int sig)
{
    int save_errno = errno ;
    int msg = sig;
    send(pipefd[1], (char*) &msg, 1, 0);
    errno = save_errno ;
}
ret = socketpair(PF_UNIX, SOCK_STREAM, 0, pipefd);
/*当

I/O多路复用函数，检测到

pipefd[0]，有内容到来时，则使用

recv读取

*/
char signals[1024];
ret = recv(pipefd[0], signals, sizeof(signals), 0);
```

将socketpair的一端置于select/poll/epoll等函数的监控下，当信号到达的时候，信号处理函数会往socketpair的另一端写入1个字节，即信号的

值。此时，主程序的select/poll/epoll函数就能侦测到此事，对socketpair执行recv操作，获取到信号处理函数写入的信号值，进行相应的处理。

6.14 总结

Linux的signal机制是一种原始的进程间通信机制，传递的信息有限，很难传递复杂的消息，加上信号处理函数和进程处于两条执行逻辑流，会带来函数的重入问题，因此signal机制不适合作为进程间通信的主要手段。但是信号又不是完全无用的，对于某些不频繁发生的异步事件，还是可以使用signal来通知进程。

第7章 理解Linux线程（1）

相对于Unix操作系统40多年的光辉历史，线程算是出现得比较晚的。在20世纪90年代线程才慢慢流行起来，而POSIX threads标准的确立已经是1995年的事情了。

Unix原本是不支持线程的，线程概念的引入给Unix家族带来了一些麻烦，很多函数都不是线程安全（thread-safe）的，需要重新定义，信号机制在线程加入以后也变得更加复杂了。

在单核CPU时代，多线程的需求并没有那么强烈，但是随着时间的流逝，事情发生了变化。2005年3月，Herb Sutter在Dobb's Journal上发表了《The Free Lunch is over: A Fundamental Turn Toward Concurrency in Software》一文，文章分析处理器厂商改善CPU性能的传统方法，如提升时钟速度和指令吞吐量，基本已经走到了尽头，处理器开始向超线程和多核架构靠拢，多核的时代已然来临。为了让代码运行得更快，单纯地依赖更快的硬件已经无法满足要求。程序员需要编写并发代码，以便充分发挥多核处理器的强大功能，并且使程序的性能得到提升。

7.1 线程与进程

在Linux下，程序或可执行文件是一个静态的实体，它只是一组指令的集合，没有执行的含义。进程是一个动态的实体，有自己的生命周期。线程是操作系统进程调度器可以调度的最小执行单元。进程和线程的关系如图7-1所示。

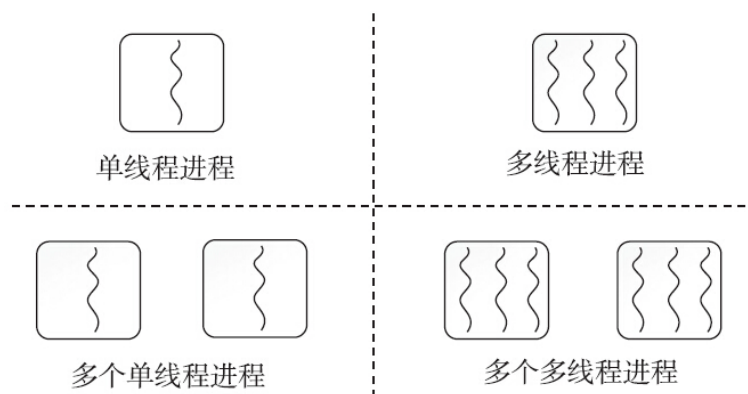


图7-1 线程和进程的关系

一个进程可能包含多个线程，传统意义上的进程，不过是多线程的一种特例，即该进程只包含一个线程。

为什么要有多线程？

举个生活中的例子，这就好比去银行办理业务。到达银行后，首先找到领导的机器领取一个号码，然后坐下来安心等待。这时候你一定希望，办理业务的窗口越多越好。如果把整个营业大厅当成一个进程的话，那么每一个窗口就是一个工作线程。

这种场景在Linux中屡见不鲜。编程的思想（如图7-2所示）和生活中解决问题的想法总是类似的。

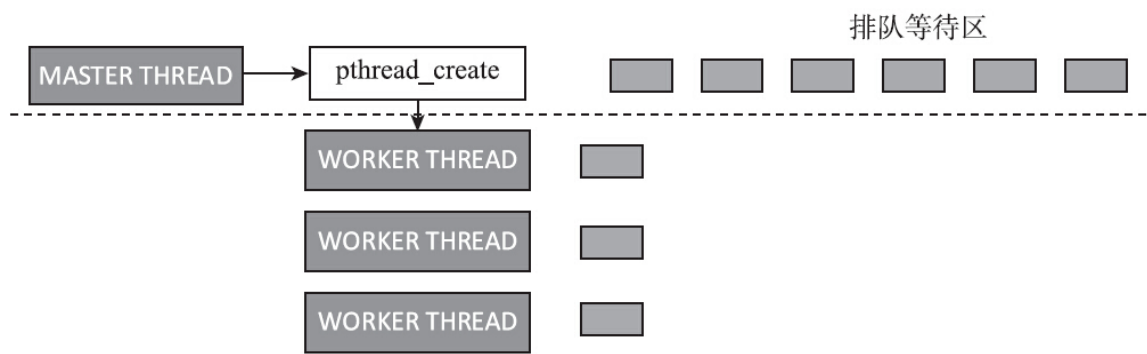


图7-2 Master-Worker并发模型

有人说不必非要使用线程，多个进程也能做到这点。的确如此。Unix/Linux原本的设计是没有线程的，类Unix系统包括Linux从设计上更倾向于使用进程，反倒是Windows因为创建进程的开销巨大，而更加钟爱线程。

那么线程是不是一种设计上的冗余呢？其实不是这样的。

进程之间，彼此的地址空间是独立的，但线程会共享内存地址空间（如图7-3所示）。同一个进程的多个线程共享一份全局内存区域，包括初始化数据段、未初始化数据段和动态分配的堆内存段。

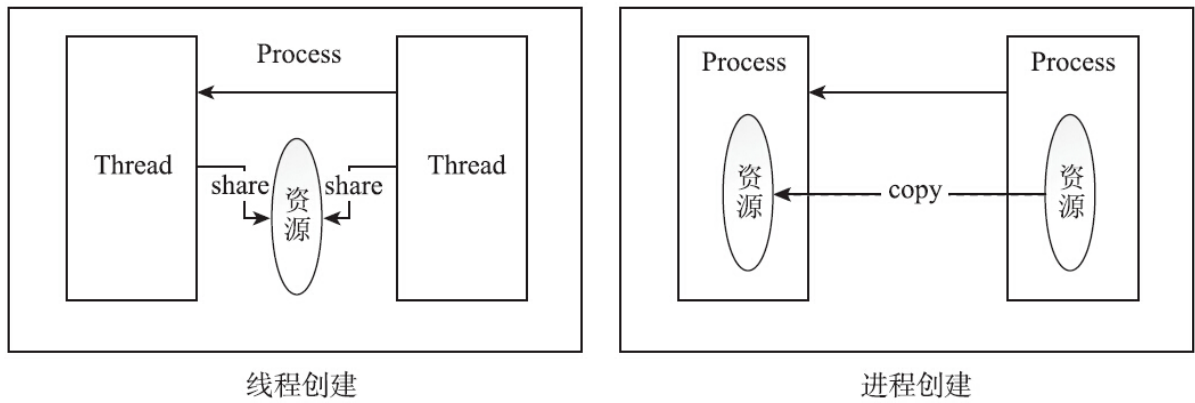


图7-3 线程之间共享资源

这种共享给线程带来了很多的优势：

- 创建线程花费的时间要少于创建进程花费的时间。
- 终止线程花费的时间要少于终止进程花费的时间。
- 线程之间上下文切换的开销，要小于进程之间的上下文切换。
- 线程之间数据的共享比进程之间的共享要简单。

下面用一个简单的实验，来比较下创建10万个进程和10万个线程各自的开销。

创建进程的测试程序将会执行如下操作：

- 1) 调用fork函数创建子进程，子进程无实际操作，调用exit函数立刻退出，父进程等待子进程退出。
- 2) 重复执行步骤1，共执行10万次。

创建线程的测试程序则执行如下操作：

- 1) 调用pthread_create创建线程，线程无实际操作；调用pthread_exit函数，立刻退出；主线程调用pthread_join函数等待线程退出。
- 2) 重复执行步骤1，共执行10万次。

服务器CPU的情况为：Intel 2.1GHz Xeon E5-2620（24核），下面看下测试结果，见表7-1。

表7-1 创建线程或子进程之前无内存分配，程序耗时比较

进 程 测 试			线 程 测 试		
real	user	sys	real	user	sys
9.598s	0.140s	10.327s	2.128s	0.512s	1.805s

从测试结果上看，创建线程花费的时间约是创建进程花费时间的五分之一。

在上述测试中，调用fork函数和pthread_create函数之前，并没有分配大块内存。一旦分配大块内存，考虑到创建进程需要拷贝页表，而创建线程不需要，则两者之间效率上的差距会进一步拉大，见表7-2。

表7-2 创建线程或子进程之前，堆上分配了40MB空间

进 程 测 试			线 程 测 试		
real	user	sys	real	user	sys
100.631s	0.502s	101.898s	2.304s	0.567s	1.926s

线程间的上下文切换，指的是同一个进程里不同线程之间发生的上下文切换。由于线程原本属于同一个进程，它们会共享地址空间，大量资源共享，切换的代价小于进程之间的切换是自然而然的事情。

线程之间通信的代价低于进程之间通信的代价。从生活的角度来类比，部门内的协作总是要比跨部门的协作来得顺溜。线程共享地址空间的设计，让多个线程之间的通信变得非常简单。一个进程内的多个线程，就像一个软件研发小组内部的不同员工，共享代码、服务器、打印机、资料，彼此之间有分工协作，沟通协作成本比较低。进程之间的通信代价则要高很多。进程之间不得不采用一些进程间通信的手段（如管道、共享内存及信号量等）来协作。

前面是从操作系统的角度来分析线程优势的，从用户或应用的视角来分析，多线程的程序也有很多的优势。

1.发挥多核优势，充分利用CPU资源

CPU是一种资源，如果一方面CPU资源大量闲置，处于IDLE的状态，另一方面很多任务得不到及时的处理，处于排队等待的状态，这就表明资源没有得到有效的利用，本质上是一种浪费。

可以想象如下场景：你在火车站买票，10个售票窗口，有9个窗口的售票员暂停服务，但是这9个售票员却在嗑瓜子，玩手机，大厅里排队者有几百人。

你排在最后！！！！

你是不是很愤怒。是的，编程领域也一样，如果存在多个相同的任务，彼此之间并行不悖，互不依赖（或者依赖性很小），那么启动多个线程并发处理，是一个不错的选择（如图7-4所示）。虽然对每个任务而言，处理的时间并没有缩短，但是在相同时间内，处理了更多的任务。

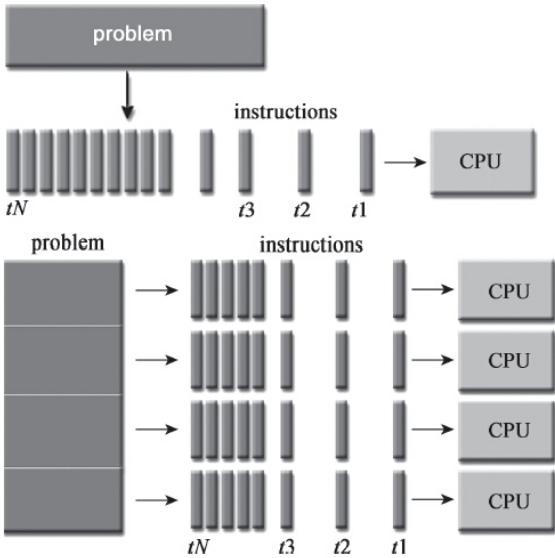


图7-4 并发执行，充分利用CPU资源

2.更自然的编程模型

有很多程序，天生就适合用多线程。将工作切分成多个模块，并为每个模块分配一个或多个执行单元，更符合人类解决问题的思路。

以文本编辑程序为例，用户的输入需要及时响应，必须要有线程来监控鼠标和键盘；如果用户删除了第一页的某一行，后面很多页的格式都会受到影响，这时就需要有文本格式化线程在后台执行格式处理；很多文本编辑软件都有自动保存的功能，第三个线程会周期性地将文件内容写入磁盘；很多文本编辑软件都有检测拼写错误的功能，或许我们需要第四个线程……

上述的分工是很自然的事情，想象一下如果将所有工作都放在一个单线程的进程里面，那么该进程是不是就不得不处理庞杂而又繁芜的事情？程序结构也就会变得异常复杂。

没有银弹。多线程带来优势的同时，也存在一些弊端。

1) 多线程的进程，因地址空间的共享让该进程变得更加脆弱

多个线程之中，只要有一个线程不够健壮存在bug（如访问了非法地址引发的段错误），就会导致进程内的所有线程一起完蛋。正所谓：

覆巢之下，安有完卵

城门失火，殃及池鱼

相比之下，进程的地址空间互相独立，彼此隔离得更加彻底。多个进程之间互相协同，一个进程存在bug导致异常退出，不会影响到其他进程。

2) 线程模型作为一种并发的编程模型，效率并没有想象的那么高，会出现复杂度高、易出错、难以测试和定位的问题

目前存在的并发编程，基本可以分成两类：

·共享状态式

·消息传递式

线程模型采用的是第一种。从现在开始，停止幻想，欢迎来到真实的世界。

一个程序员碰到了一个问题，他决定用多线程来解决。现在两个他问题了有 [1]。

——关于线程的冷笑话

在真实的场景中，多线程编程是很复杂的。前面所说的多个任务并行不悖，互不依赖，在大多数情况下只是一种美好的幻想。

首先，多个线程之间，存在负载均衡的问题，现实中很难将全部任务等分给每个线程。想象一下，如果存在10个线程，一个线程承担了90%的任务，9个线程承担了10%的任务，整体的效率立刻就降了下来。

有人说，怎么会有这么愚蠢的设计呢。试想如下场景：你需要用支持10个并发线程的服务器去计算 $1 \sim 10^{10}$ 以内的所有素数，要怎么设计？首先进入脑海的第一反应是不是将 $1 \sim 10^{10}$ 这个范围平均分成10份，每一份有 10^9 个数，10个线程分别查找范围内的素数？这就是糟糕的设计，尽管每个线程负责的范围是相同的，但是每个线程的负载并不均匀，因为判断一个较大的数是不是素数，通常要比判断较小的数所花费的时间更长。当然这个例子有比较妥善的解决方案，但是在很多情况下，很难将负载均匀地分配给各个线程。

其次，多个线程的任务之间还可能存在顺序依赖的关系，一个线程未能完成某些操作之前，其他线程不能或不应该运行。

多个线程之间需要同步。想象如下场景：你和你的朋友合租一套公寓，这套公寓只有1个卫生间。当你朋友正在使用卫生间的时候，你就无法使用了。多线程也会遇到类似的问题。多个线程生活在进程地址空间这同一个屋檐下，若存在多个线程操作共享资源，则需要同步，否则可能会出现结果错误、数据结构遭到破坏甚至是程序崩溃等后果。因此多线程编程中存在临界区的概念，临界区的代码只允许一个线程执行，线程提供了锁机制来保护临界区。当其他线程来到临界区却无法申请到锁时，就可能陷入阻塞，不再处于可执行状态，线程可能不得不让出CPU资源。如果设计不合理，临界区非常多，线程之间的竞争异常激烈，频繁地上下文切换也会导致性能急剧恶化。

上面两种情况的存在，决定了多线程并非总是处于并发的状态，加速也并非线性的。4个工作线程未必能带来4倍的效率，加速比取决于可以串行执行的部分在全部工作中所占的比例。

有人曾经这样打比方：多进程属于立体交通系统，虽然造价高，上坡下坡比较耗油，但是堵车少；多线程属于平面交通系统，造价低，但是红绿灯太多，老堵车。个人觉得这个比方是很有道理的。

多线程模型的复杂度更是不容小觑。很多人诟病多线程模型，就在于它不符合人的心智模型。俗语道，一心不可两用，人很难同时控制多条走走停停，彼此又有交互和同步的控制流。由于进程调度的无序性，严格来说多线程程序的每次执行其实并不一样，很难穷举所有的时序组合，所以我们永远无法宣称多线程的程序经过了充分的测试。在某些特殊时序的条件下，bug可能会出现，这种bug难以复现，而且难以排查。所以编程时，需要谨慎地设计，以确保程序能够在所有的时序条件下正常运行。

对于多线程编程，还存在四大陷阱，一不小心就可能落入陷阱之中。这四个陷阱分别是：

·死锁（Dead Lock）

- 饿死 (Starvation)

- 活锁 (Live Lock)

- 竞态条件 (Race Condition)

客观地说，多线程编程的难度要更大一些，需要程序员更加小心，更加谨慎。当你需要使用多线程的时候，一定要花费足够的时间小心地规划每个线程的分工，尽可能地减少线程之间的依赖。良好的设计，合理的分工是多线程编程至关重要的环节。若初期随意地设计线程的分工，那么在最后，你很有可能不得不花费大量的时间来优化性能，定位bug，甚至不得不推倒重来。

[1] 此处的语序混乱是故意的，暗讽由于线程、多条控制流、时序失去控制，导致混乱。

7.2 进程ID和线程ID

在Linux中，目前的线程实现是Native POSIX Thread Library，简称NPTL。在这种实现下，线程又被称为轻量级进程（Light Weighted Process），每一个用户态的线程，在内核之中都对应一个调度实体，也拥有自己的进程描述符（task_struct结构体）。

没有线程之前，一个进程对应内核里的一个进程描述符，对应一个进程ID。但是引入了线程的概念之后，情况就发生了变化，一个用户进程下管辖N个用户态线程，每个线程作为一个独立的调度实体在内核态都有自己的进程描述符，进程和内核的进程描述符一下子就变成了1：N的关系，POSIX标准又要求进程内的所有线程调用getpid函数时返回相同的进程ID。如何解决上述问题呢？

内核引入了线程组（Thread Group）的概念。

```
struct task_struct {...
    pid_t pid;
    pid_t tgid
    ...
    struct task_struct *group_leader;
    ...
    struct list_head thread_group;
    ...
}
```

多线程的进程，又被称为线程组，线程组内的每一个线程在内核之中都存在一个进程描述符（task_struct）与之对应。进程描述符结构体中的pid，表面上看对应的是进程ID，其实不然，它对应的是线程ID；进程描述符中的tgid，含义是Thread Group ID，该值对应的是用户层面的进程ID，具体见表7-3。

表7-3 线程ID和进程ID的值

用 户 态	系 统 调 用	内核进程描述符中对应的结构
线程 ID	pid_t gettid(void);	pid_t pid
进程 ID	pid_t getpid(void);	pid_t tgid

本节介绍的线程ID，不同于后面会讲到的pthread_t类型的线程ID，和进程ID一样，线程ID是pid_t类型的变量，而且是用来唯一标识线程的一个整型变量。那么如何查看一个线程的ID呢？

```
manu@manu-hacks:~$ ps -eLf
...
UID      PID  PPID  LWP  C  NLWP  STIME  TTY          TIME CMD
syslog   837    1   837  0    4  22:20 ?        00:00:00 rsyslogd
syslog   837    1   838  0    4  22:20 ?        00:00:00 rsyslogd
syslog   837    1   839  0    4  22:20 ?        00:00:00 rsyslogd
syslog   837    1   840  0    4  22:20 ?        00:00:00 rsyslogd
...
```

ps命令中的-L选项，会显示出线程的如下信息。

- LWP：线程ID，即gettid（）系统调用的返回值。
- NLWP：线程组内线程的个数。

所以从上面可以看出rsyslogd进程是多线程的，进程ID为837，进程内有4个线程，线程ID分别为837、838、839和840（如图7-5所示）。

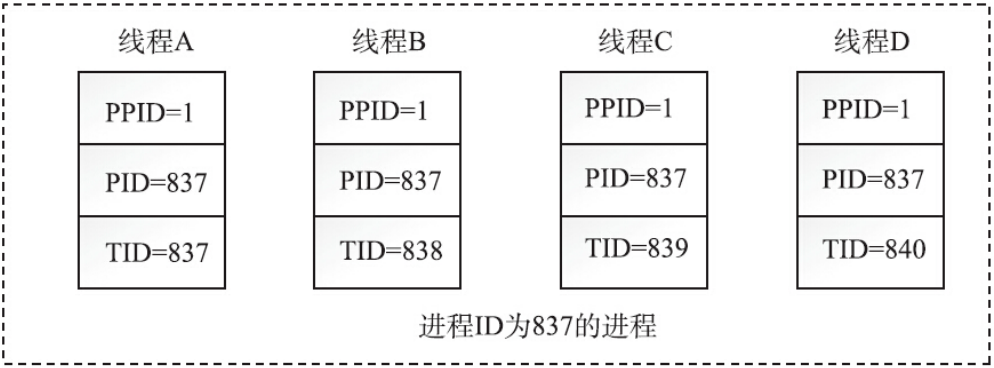


图7-5 进程ID和线程ID（调度域）

已知某进程的进程ID，该如何查看该进程内线程的个数及其线程ID呢？其实可以通过/proc/PID/task/目录下的子目录来查看，如下。因为procs在task下会给进程的每个线程建立一个子目录，目录名为线程ID。

```
manu@manu-hacks:~$ ll /proc/837/task/总用量
dr-xr-xr-x 6 syslog syslog 0 4月

16 22:32 ./
dr-xr-xr-x 9 syslog syslog 0 4月

16 22:20 ../
dr-xr-xr-x 6 syslog syslog 0 4月

16 22:32 837/
dr-xr-xr-x 6 syslog syslog 0 4月

16 22:32 838/
dr-xr-xr-x 6 syslog syslog 0 4月

16 22:32 839/
dr-xr-xr-x 6 syslog syslog 0 4月

16 22:32 840/
```

对于线程，Linux提供了gettid系统调用来返回其线程ID，可惜的是glibc并没有将该系统调用封装起来，再开放出接口来供程序员使用。如果确实需要获取线程ID，可以采用如下方法：

```
#include <sys/syscall.h>
int TID = syscall(SYS_gettid);
```

从上面的示例来看，rsyslogd是个多线程的进程，进程ID为837，下面有一个线程的ID也是837，这不是巧合。线程组内的第一个线程，在用户态被称为主线程（main thread），在内核中被称为Group Leader。内核在创建第一个线程时，会将线程组ID的值设置成第一个线程的线程ID，group_leader指针则指向自身，即主线程的进程描述符，如下。

```
/*线程组

ID等于主线程的

ID.

group_leader指向自身

*/
p->tgid = p->pid;
p->group_leader = p;
INIT_LIST_HEAD(&p->thread_group);
```

所以可以看到，线程组内存在一个线程ID等于进程ID，而该线程即为线程组的主线程。

至于线程组其他线程的ID则由内核负责分配，其线程组ID总是和主线程的线程组ID一致，无论是主线程直接创建的线程，还是创建出来的线程再次创建的线程，都是这样。

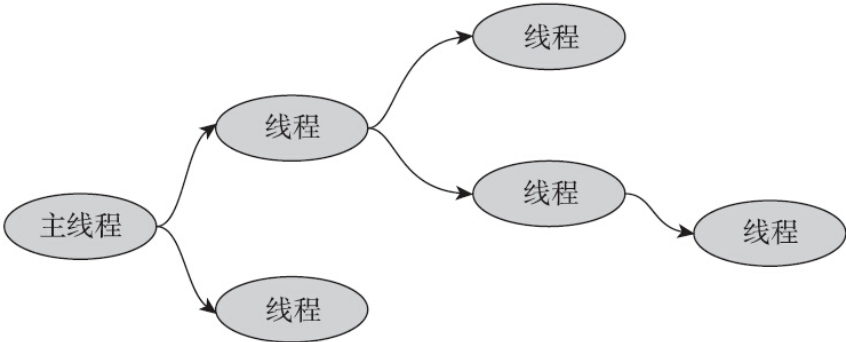
```
if (clone_flags & CLONE_THREAD)
    p->tgid = current->tgid;
if (clone_flags & CLONE_THREAD) {
    p->group_leader = current->group_leader;
    list_add_tail_rcu(&p->thread_group, &p->group_leader->thread_group);
}
```

通过group_leader指针，每个线程都能找到主线程。主线程存在一个链表头，后面创建的每一个线程都会链入到该双向链表中。

利用上述的结构，每个线程都可以轻松地找到其线程组的主线程（通过group_leader指针），另一方面，通过线程组的主线程，也可以轻松地遍历其所有的组内线程（通过链表）。

需要强调的一点是，线程和进程不一样，进程有父进程的概念，但在线程组里面，所有的线程都是对等的关系（如图7-6所示）。

- 并不是只有主线程才能创建线程，被创建出来的线程同样可以创建线程。
- 不存在类似于fork函数那样的父子关系，大家都归属于同一个线程组，进程ID都相等，group_leader都指向主线程，而且各有各的线程ID。
- 并非只有主线程才能调用pthread_join连接其他线程，同一线程组内的任意线程都可以对某线程执行pthread_join函数。
- 并非只有主线程才能调用pthread_detach函数，其实任意线程都可以对同一线程组内的线程执行分离操作。



同一线程组的线程，没有层次关系

图7-6 线程的对等关系

7.3 pthread库接口介绍

1995年，POSIX.1c标准对POSIX线程API进行了标准化，这就是我们今天看到的pthread库的接口。

这些接口包括线程的创建、退出、取消和分离，以及连接已经终止的线程，互斥量，读写锁，线程的条件等待等（如表7-4所示）。

表7-4 POSIX线程库的接口

POSIX 函数	函数功能描述
pthread_create	创建一个线程
pthread_exit	退出线程
pthread_self	获取线程 ID
pthread_equal	检查两个线程 ID 是否相等
pthread_join	等待线程退出
pthread_detach	设置线程状态为分离状态
pthread_cancel	线程的取消（将于第 8 章介绍）
pthread_cleanup_push pthread_cleanup_pop	线程退出，清理函数注册和执行

上面提到的函数列表，是pthread的基本接口，接下来的章节，将分别介绍这些接口。

7.4 线程的创建和标识

首先要介绍的接口是创建线程的接口，即`pthread_create`函数。程序开始启动的时候，产生的进程只有一个线程，我们称之为主线程或初始线程。对于单线程的进程而言，只存在主线程一个线程。如果想在主线程之外，再创建一个或多个线程，就需要用到这个接口了。

7.4.1 pthread_create函数

pthread库提供了如下接口来创建线程：

```
#include <pthread.h>
int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void*),
                  void *restrict arg);
```

pthread_create函数的第一个参数是pthread_t类型的指针，线程创建成功的话，会将分配的线程ID填入该指针指向的地址。线程的后续操作将使用该值作为线程的唯一标识。

第二个参数是pthread_attr_t类型，通过该参数可以定制线程的属性，比如可以指定新建线程栈的大小、调度策略等。如果创建线程无特殊的要求，该值也可以是NULL，表示采用默认属性。

第三个参数是线程需要执行的函数。创建线程，是为了让线程执行一定的任务。线程创建成功之后，该线程就会执行start_routine函数，该函数之于线程，就如同main函数之于主线程。

第四个参数是新建线程执行的start_routine函数的入参。

新建线程如果想要正常工作，则可能需要入参，那么主线程在调用pthread_create的时候，就可以将入参的指针放入第四个参数以传递给新建线程。

如果线程的执行函数start_routine需要很多入参，传递一个指针就能提供足够的信息吗？答案是能。线程创建者（一般是主线程）和线程约定一个结构体，创建者便把信息填入该结构体，再将结构体的指针传递给子进程，子进程只要解析该结构体，就能取出需要的信息。

如果成功，则pthread_create返回0；如果不成功，则pthread_create返回一个非0的错误码。常见的错误码如表7-5所示。

表7-5 pthread_create的错误码及描述

返回 值	描 述
EAGAIN	系统资源不够，或者创建线程的个数超过系统对一个进程中线程总数的限制
EINVAL	第二个参数 attr 值不合法
EPERM	没有合适的权限来设置调度策略或参数

pthread_create函数的返回情况有些特殊，通常情况下，函数调用失败，则返回-1，并且设置errno。pthread_create函数则不同，它会将errno作为返回值，而不是一个负值。

```
void * thread_worker(void *)
{
    printf("

I am thread worker"

);
    pthread_exit(NULL)
}
pthread_t tid ;
int ret= 0;
ret = pthread_create(&tid,NULL,&thread_worker,NULL);
if (ret != 0)/* 注意此处，不能用

ret < 0 作为出错判断

*/
{
    /*ret is the errno*/
    /*error handler*/
}
```

7.4.2 线程ID及进程地址空间布局

`pthread_create`函数，会产生一个线程ID，存放在第一个参数指向的地址中。该线程ID和7.2节分析的线程ID是一回事吗？

答案是否定的。

7.2节提到的线程ID，属于进程调度的范畴。因为线程是轻量级进程，是操作系统调度器的最小单位，所以需要数值来唯一标识该线程。

`pthread_create`函数产生并记录在第一个参数指向地址的线程ID中，属于NPTL线程库的范畴，线程库的后续操作，就是根据该线程ID来操作线程的。

线程库NPTL提供了`pthread_self`函数，可以获取到线程自身的ID：

```
#include <pthread.h>
pthread_t pthread_self(void);
```

在同一个线程组内，线程库提供了接口，可以判断两个线程ID是否对应着同一个线程：

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

返回值是0的时候，表示两个线程是同一个线程，非零值则表示不是同一个线程。

`pthread_t`到底是个什么样的数据结构呢？因为POSIX标准并没有限制`pthread_t`的数据类型，所以该类型取决于具体实现。对于Linux目前使用的NPTL实现而言，`pthread_t`类型的线程ID，本质就是一个进程地址空间上的一个地址。

是时候看一下进程地址空间的布局了。在x86_64平台上，用户地址空间约为128TB，对于地址空间的布局，系统有如下控制选项：

```
cat /proc/sys/vm/legacy_va_layout
0
```

该选项影响地址空间的布局，主要是影响`mmap`区域的基地址位置，以及`mmap`是向上还是向下增长。如果该值为1，那么`mmap`的基地址`mmap_base`变小（约在128T的三分之一处），`mmap`区域从低地址向高地址扩展。如果该值为0，那么`mmap`区域的基地址在栈的下面（约在128T空间处），`mmap`区域从高地址向低地址扩展。默认值为0，布局如图7-7所示。

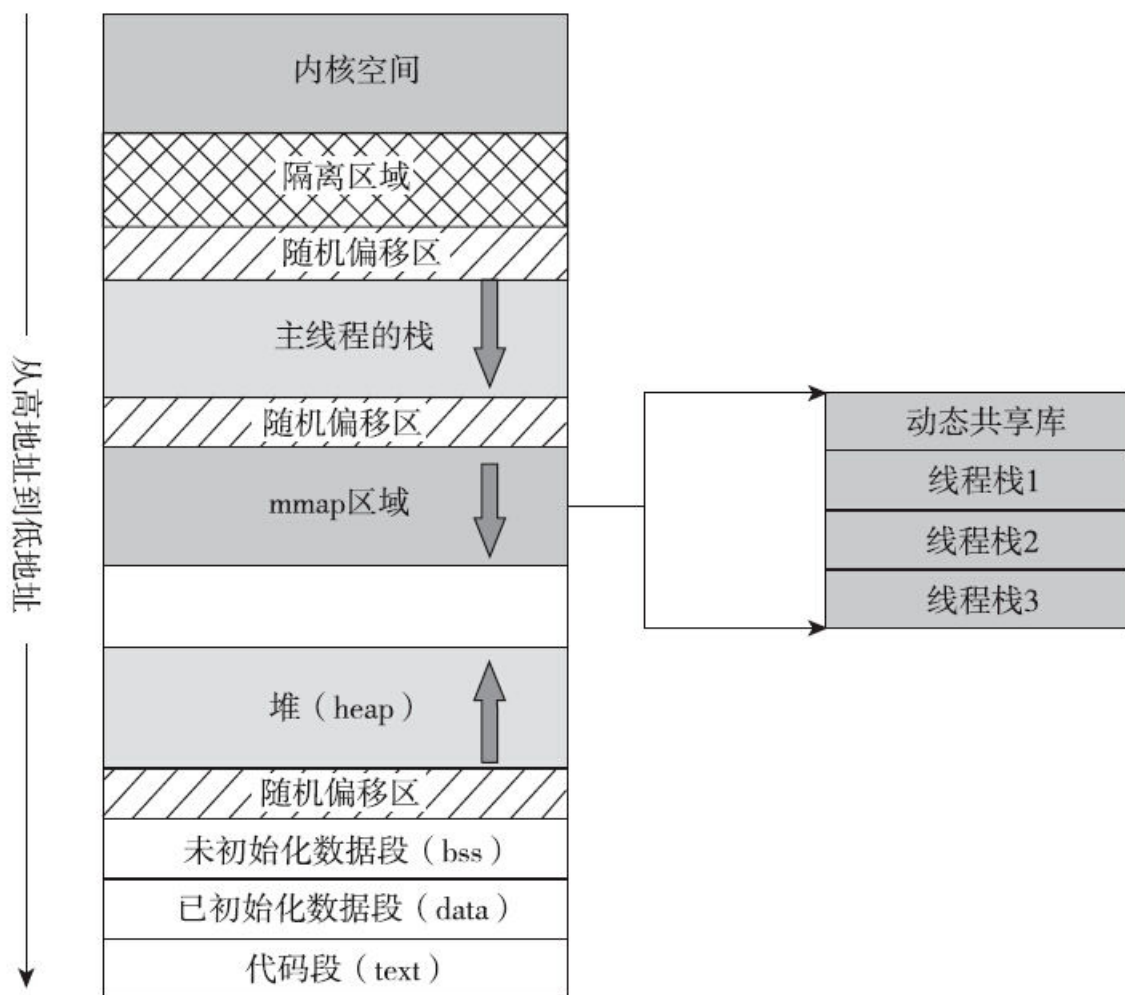


图7-7 多线程进程的地址空间

可以通过procfs或pmap命令来查看进程的地址空间的情况：

```
pmap PID
```

或者

```
cat /proc/PID/maps
```

在接近128TB的巨大地址空间里面，代码段、已初始化数据段、未初始化数据段，以及主线程的栈，所占用的空间非常小，都是KB、MB这个数量级的，如下：

```
manu@manu-hacks:~$ pmap 3706
3706: ./process_map
0000000000400000      4K r-x-- process_map
0000000000601000      4K r---- process_map
0000000000602000      4K rw--- process_map...

00007ffdd5f68000   5128K rw--- [ stack ] /*栈在
```

128T位置附近

*/

由于主线程的栈大小并不是固定的，要在运行时才能确定大小（上限大概在8MB左右），因此，在栈中不能存在巨大的局部变量，另外编写递归函数时一定要小心，递归不能太深，否则很可能耗尽栈空间。

如下面的例子所示，无尽地递归，很轻易就耗尽了栈的空间：

```
int i = 0;
void func()
{
    int buffer[256];
    printf("i = %d\n", i);
    i++;
    func();
}
int main()
{
    func();
    sleep(100);
}
```

上面代码的递归永不停息，每次递归，都会消耗约1KB（256个int型为1KB）的栈空间。通过运行可以看出，主线程栈最大也就在8MB左右：

```
i = 8053
i = 8054
i = 8055段错误
```

（核心已转储）

进程地址空间之中，最大的两块地址空间是内存映射区域和堆。堆的起始地址特别低，向上扩展，mmap区域的起始地址特别高，向下扩展。

用户调用pthread_create函数时，glibc首先要为线程分配线程栈，而线程栈的位置就落在mmap区域。glibc会调用mmap函数为线程分配栈空间。pthread_create函数分配的pthread_t类型的线程ID，不过是分配出来的空间里的一个地址，更确切地说是一个结构体的指针，如图7-8所示。

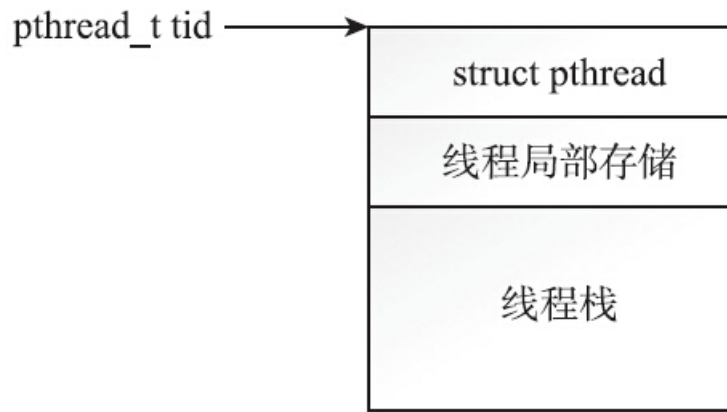


图7-8 线程ID的本质是内存地址

创建两个线程，将其pthread_self()的返回值打印出来，输出如下：

```
address of tid in thread-1 = 0x7f011ca12700
address of tid in thread-2 = 0x7f011c211700
```

线程ID是进程地址空间内的一个地址，要在同一个线程组内进行线程之间的比较才有意义。不同线程组内的两个线程，哪怕两者的pthread_t值是一样的，也不是同一个线程，这是显而易见的。

很有意思的一点是，pthread_t类型的线程ID很有可能会被复用。在满足下列条件时，线程ID就有可能被复用：

- 1) 线程退出。
- 2) 线程组的其他线程对该线程执行了pthread_join，或者线程退出前将分离状态设置为已分离。
- 3) 再次调用pthread_create创建线程。

为什么pthread_t类型的线程ID会被复用，这点将在后面进行分析。下面通过测试来证明一下：

/*省略了

```
error handler*/
void* thread_work(void* param)
{
    int TID = syscall(SYS_gettid);
    printf("thread-%d: gettid return %d\n",TID,TID);
    printf("thread-%d: pthread_self return %p\n",TID,(void *)pthread_self());
    printf("thread-%d: I will exit now\n",TID);
    pthread_exit(NULL);
    return NULL;
}
int main(int argc ,char* argv[])
{
    pthread_t tid = 0;
    int ret
    ret = pthread_create(&tid,NULL,thread_work,NULL);
    ret = pthread_join(tid,NULL);
    ret = pthread_create(&tid,NULL,thread_work,NULL);
    ret = pthread_join(tid,NULL);
    return 0;
}
```

输出结果如下：

```
thread-4158: gettid return 4158
thread-4158: pthread_self return 0x7f43a27d0700
thread-4158: I will exit now
thread-4159: gettid return 4159
thread-4159: pthread_self return 0x7f43a27d0700
thread-4159: I will exit now
```

从输出结果上看，对于pthread_t类型的线程ID，虽然在同一时刻不会存在两个线程的ID值相同，但是如果线程退出了，重新创建的线程很可能复用了同一个pthread_t类型的ID。从这个角度看，如果要设计调试日志，用pthread_t类型的线程ID来标识进程就不太合适了。用pid_t类型的线程ID则是一个比较不错的选择。

```
#include <sys/syscall.h>
int TID = syscall(SYS_gettid);
```

采用pid_t类型的线程ID来唯一标识进程有以下优势：

- 返回类型是pid_t类型，进程之间不会存在重复的线程ID，而且不同线程之间也不会重复，在任意时刻都是全局唯一的值。
- procfs中记录了线程的相关信息，可以方便地查看/proc/pid/task/tid来获取线程对应的信息。
- ps命令提供了查看线程信息的-L选项，可以通过输出中的LWP和NLWP，来查看同一个线程组的线程个数及线程ID的信息。

另外一个比较有意思的功能是我们可以给线程起一个有意义的名字，命名以后，既可以从procfs中获取到线程的名字，也可以从ps命令中得到线程的名字，这样就可以更好地辨识不同的线程。

Linux提供了prctl系统调用：

```
#include <sys/prctl.h>
int prctl(int option, unsigned long arg2,
          unsigned long arg3, unsigned long arg4,
          unsigned long arg5)
```

这个系统调用和ioctl非常类似，通过option来控制系统调用的行为。当需要给线程设定名字的时候，只需要将option设为PR_SET_NAME，同时将线程的名字作为arg2传递给prctl系统调用即可，这样就能给线程命名了。

下面是示例代码：

```
void thread_setnamev(const char* namefmt, va_list args)
{
    char name[17];
```

```
    vsnprintf(name, sizeof(name), namefmt, args);
    prctl(PR_SET_NAME, name, NULL, NULL, NULL);
}
void thread_setname(const char* namefmt, ...)
{
    va_list args;
    va_start(args, namefmt);
    thread_setnamev(namefmt, args);
    va_end(args);
}
thread_setname("BEAN-%d", num);
```

这里共创建了四个线程，按照调用`pthread_create`的顺序，将0、1、2、3作为参数传递给线程，然后调用`prctl`给每个线程起名字：分别为BEAN-0、BEAN-1、BEAN-2和BEAN-3。命名以后可以通过`ps`命令来查看线程的名字：

```
manu@manu-hacks:~$ ps -L -p 3454
  PID   LWP  TTY      TIME  CMD
 3454   3454 pts/0    00:00:00 pthread_tid
 3454   3455 pts/0    00:00:00 BEAN-0
 3454   3456 pts/0    00:00:00 BEAN-1
 3454   3457 pts/0    00:00:00 BEAN-2
 3454   3458 pts/0    00:00:00 BEAN-3
manu@manu-hacks:~$ cat /proc/3454/task/3457/status
Name:      BEAN-2
State:     S (sleeping)
Tgid:      3454
```

这是一个很有用的技巧。给线程命了名，就可以很直观地区分各个线程，尤其是在线程比较多，且其分工不同的情况下。

7.4.3 线程创建的默认属性

线程创建的第二个参数是pthread_attr_t类型的指针，pthread_attr_init函数会将线程的属性重置成默认值。

```
pthread_attr_t attr;
pthread_attr_init(&attr);
```

在创建线程时，传递重置过的属性，或者传递NULL，都可以创建一个具有默认属性的线程，见表7-6。

表7-6 线程的属性及默认值

属 性	默 认 值	说 明
contentionscope	PTHREAD_SCOPE_SYSTEM	进程调度相关，NPTL 实现中，线程只支持在操作系统范围内竞争 CPU 资源
detachstate	PTHREAD_CREATE_JOINABLE	可分离状态，详情请见 pthread_join 章节（7.6.1 节）
stackaddr	NULL	不指定线程栈的基址，由系统决定栈基址
stacksize	8196(KB)	默认线程栈大小为 8MB（ulimit -s 查看）
guardsize	PAGESIZE	警戒缓冲区
priority	0	进程调度相关，优先级为 0
policy	SCHED_OTHER	进程调度相关，调度策略为 SCHED_OTHER
inheritsched	PTHREAD_INHERIT_SCHED	进程调度相关，继承启动进程的调度策略

手册给出了一个如何展示线程属性的例子，若你需要展示线程的属性，则可以参考手册。

本节现在来介绍线程栈的基地址和大小。默认情况下，线程栈的大小为8MB：

```
manu@manu-hacks:~$ ulimit -s
8192
```

调用pthread_attr_getstack函数可以返回线程栈的基地址和栈的大小。出于可移植性的考虑不建议指定线程栈的基地址。但是有时候会有修改线程栈的大小的需要。

一个线程需要分配8MB左右的栈空间，就决定了不可能无限地创建线程，在进程地址空间受限的32位系统里尤为如此。在32位系统下，3GB的用户地址空间决定了能创建线程的个数不会太多。如果确实需要很多的线程，可以调用接口来调整线程栈的大小：

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr,
                             size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr,size_t *stacksize);
```

7.5 线程的退出

有生就有灭，线程执行完任务，也需要终止。

下面的三种方法中，线程会终止，但是进程不会终止（如果线程不是进程组里的最后一个线程的话）：

- 创建线程时的start_routine函数执行了return，并且返回指定值。
- 线程调用pthread_exit。
- 其他线程调用了pthread_cancel函数取消了该线程（详见第8章）。

如果线程组中的任何一个线程调用了exit函数，或者主线程在main函数中执行了return语句，那么整个线程组内的所有线程都会终止。

值得注意的是，pthread_exit和线程启动函数（start_routine）执行return是有区别的。在start_routine中调用的任何层级的函数执行pthread_exit（）都会引发线程退出，而return，只能是在start_routine函数内执行才能导致线程退出。

```
void* start_routine(void* param)
{
    ...

    foo();
    bar();
    return NULL;
}
void foo()
{
    ...
    pthread_exit(NULL);
}
```

如果foo函数执行了pthread_exit函数，则线程会立刻退出，后面的bar就会没有机会执行了。

下面来看看pthread_exit函数的定义：

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

value_ptr是一个指针，存放线程的“临终遗言”。线程组内的其他线程可以通过调用pthread_join函数接收这个地址，从而获取到退出线程的临终遗言。如果线程退出时没有什么遗言，则可以直接传递NULL指针，如下所示：

```
pthread_exit(NULL);
```

但是这里有一个问题，就是不能将遗言存放到线程的局部变量里，因为如果用户写的线程函数退出了，线程函数栈上的局部变量可能就不复存在了，线程的临终遗言也就无法被接收者读到，示例如下。

```
void* thread_work(void* param)
{
    int ret = -1;
    ret = whatever();
    pthread_exit(&ret);
}
```

上述用法是一种典型的错误用法，因为当线程退出时，线程栈已经不复存在了，上面的ret变量也已经无法访问了。那我们应该如何正确地传递返回值呢？

- 如果是int型的变量，则可以使用“pthread_exit((int*)ret);”。
- 使用全局变量返回。
- 将返回值填入到用malloc在堆上分配的空间里。
- 使用字符串常量，如pthread_exit(“hello, world”)。

第一种是tricky的做法，我们将返回值ret进行强制类型转换，接收方再把返回值强制转换成int。但是不推荐使用这种方法。这种方法虽然是奏效的，但是太tricky，而且C标准没有承诺将int型转成指针后，再从指针转成int型时，数据一直保持不变。

第二种方法使用全局变量，其他线程调用pthread_join时也可见这个变量。

第三种方法是用malloc，在堆上分配空间，然后将返回值填入其中。因为堆上的空间不会随着线程的退出而释放，所以pthread_join可以取出返回值。切莫忘记释放该空间，否则会引起内存泄漏。

第四种方法之所以可行，是因为字符串常量有静态存储的生存期限。

传递线程的返回值，除了pthread_exit函数可以做到，线程的启动函数（start_routine函数）return也可以做到，两者的数据类型要保持一致，都是void*类型。这也解释了为什么线程的启动函数start_routine的返回值总是void*类型，如下：

```
void pthread_exit(void *retval);
void * start_routine(void *param)
```

线程退出有一种比较有意思的场景，即线程组的其他线程仍在执行的情况下，主线程却调用

pthread_exit函数退出了。这会发生什么事情？

首先要说明的是这不是常规的做法，但是如果真的这样做了，那么主线程将进入僵尸状态，而其他线程则不受影响，会继续执行，如下。第4章曾经分析过这种场景。

```
root@newtest-1:~# ps -eL |grep thread_id
62404  62404 pts/1    00:00:00 thread_id <defunct>
62404  62405 pts/1    00:00:00 thread_id
62404  62406 pts/1    00:00:00 thread_id
```

7.6 线程的连接与分离

7.6.1 线程的连接

7.5节提到过线程退出时是可以有返回值的，那么如何取到线程退出时的返回值呢？

线程库提供了pthread_join函数，用来等待某线程的退出并接收它的返回值。这种操作被称为连接（joining）。

相关函数的接口定义如下：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

该函数第一个参数为要等待的线程的线程ID，第二个参数用来接收返回值。

根据等待的线程是否退出，可得到如下两种情况：

- 等待的线程尚未退出，那么pthread_join的调用线程就会陷入阻塞。
- 等待的线程已经退出，那么pthread_join函数会将线程的退出值（void*类型）存放到retval指针指向的位置。

线程的连接（join）操作有点类似于进程等待子进程退出的等待（wait）操作，但细细想来，还是有不同之处：

第一点不同之处是进程之间的等待只能是父进程等待子进程，而线程则不然。线程组内的成员是对等的关系，只要是在一个线程组内，就可以对另外一个线程执行连接（join）操作。如图7-9所示，线程F一样可以连接线程A。

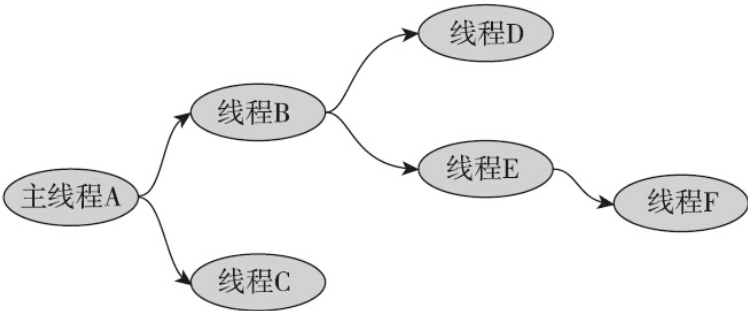


图7-9 线程的连接无等级关系

第二点不同之处是进程可以等待任一子进程的退出（用下面的代码不难做到），但是线程的连接操作没有类似的接口，即不能连接线程组内的任一线程，必须明确指明要连接的线程的线程ID。

```
wait(&status);
waitpid(-1,&status,WAIT_ANY);
```

pthread_join不能连接线程组内任意线程的做法，并不是NPTL线程库设计上的瑕疵，而是有意为之的。如果听任线程连接线程组内的任意线程，那么所谓的任意线程就会包括其他库函数私自创建的线程，当库函数尝试连接（join）私自创建的线程时，发现已经被连接过了，就会返回EINVAL错误。如果库函数需要根据返回值来确定接下来的流程，这就会引发严重的问题。正确的做法是，连接已知线程ID的那些线程，就像pthread_join函数那样。

下面来分析出错的情况，当调用失败时，和pthread_create函数一样，errno作为返回值返回。错误码的情况见表7-7。

表7-7 pthread_join的错误码和说明

返回值	说明
ESRCH	传入的线程 ID 不存在，查无此线程
EINVAL	线程不是一个可连接（joinable）的线程

返回值	说 明
EINVAL	已经有其他线程捷足先登，连接目标线程
EDEADLK	死锁，如自己连接自己，或者 A 连接 B，B 又连接 A

`pthread_join`函数之所以能够判断是否死锁和连接操作是否被其他线程捷足先登，是因为目标线程的控制结构体`struct pthread`中，存在如下成员变量，记录了该线程的连接者。

```
struct pthread *joined;
```

该指针存在三种可能，如下。

- `NULL`：线程是可连接的，但是尚没有其他线程调用`pthread_join`来连接它。
- 指向线程自身的`struct pthread`：表示该线程属于自我了断型，执行过分离操作，或者创建线程时，设置的分离属性为`PTHREAD_CREATE_DETACHED`，一旦退出，则自动释放所有资源，无需其他线程来连接。
- 指向线程组内其他线程的`struct pthread`：表示`joined`对应的线程会负责连接。

因为有了该成员变量来记录线程的连接者，所以可以判断如下场景，如图7-10所示。

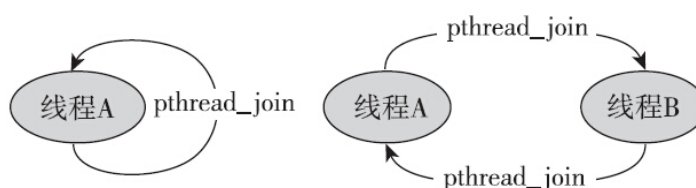


图7-10 可能返回EDEADLK的场景

不过两者还是略有区别的，第一种场景，线程A连接线程A，`pthread_join`函数一定会返回EDEADLK。但是第二种场景，大部分情况下会返回EDEADLK，不过也有例外。不管怎样，不建议两个线程互相连接。

如果两个线程几乎同时对处于可连接状态的线程执行连接操作会怎么样？

答案是只有一个线程能够成功，另一个则返回EINVAL。

NTPL提供了原子性的保证：

```
(atomic_compare_and_exchange_bool_acq(  
  
    &pd->joined, self, NULL)
```

- 如果是`NULL`，则设置成调用线程的线程ID，CAS操作（Compare And Swap）是原子操作，不可分割，决定了只有一个线程能成功。
- 如果`joined`不是`NULL`，表示该线程已经被别的线程连接了，或者正处于已分离的状态，在这两种情况下，都会返回EINVAL。

7.6.2 为什么要连接退出的线程

不连接已经退出的线程会怎么样？

如果不连接已经退出的线程，会导致资源无法释放。所谓资源指的又是什么呢？

下面通过一个测试来让事实说话。测试模拟下面两种情况：

- 主线程并不执行连接操作，待确定创建的第一个线程退出后，再创建第二个线程。
- 主线程执行连接操作，等到第一个线程退出后，再创建第二个线程。

按照时间线来发展，如图7-11所示。

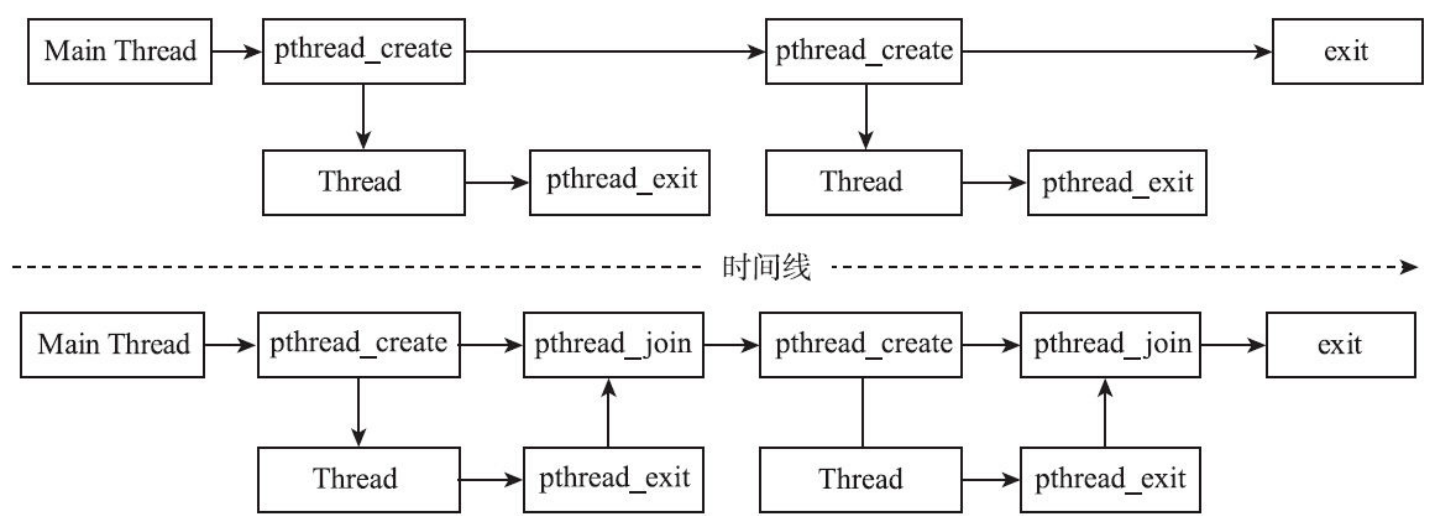


图7-11 本节代码的流程示意图

下面是代码部分，为了简化程序和便于理解，使用sleep操作来确保创建的第一个线程退出后，再来创建第二个线程。须知sleep并不是同步原语，在真正的项目代码中，用sleep函数来同步线程是不可原谅的。

```
#define GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>
#include <sys/syscall.h>
#include <sys/types.h>
#define NR_THREAD 1
#define ERRBUF_LEN 4096
void* thread_work(void* param)
{
    int TID = syscall(SYS_gettid);
    printf("thread-%d IN \n",TID);
    printf("thread-%d pthread_self return %p \n",TID,(void*)pthread_self());
    sleep(60);
    printf("thread-%d EXIT \n",TID);
    return NULL;
}
int main(int argc ,char* argv[])
{
    pthread_t tid[NR_THREAD];
    pthread_t tid_2[NR_THREAD];
    char errbuf[ERRBUF_LEN];
    int i, ret;
    for(i = 0 ; i < NR_THREAD ; i++)
    {
        ret = pthread_create(&tid[i],NULL,thread_work,NULL);
        if(ret != 0)
        {
            fprintf(stderr,"create thread failed ,return %d (%s)\n",ret,strerror_r (ret,errbuf,sizeof(errbuf)));
        }
    }
#ifdef NO_JOIN
    sleep(100);/*sleep是为了确保线程退出之后，再来重新创建线程*/
#else
    printf("join thread Begin\n");
    for(i = 0 ; i < NR_THREAD; i++)
    {
        pthread_join(tid[i],NULL);
    }
    for(i = 0 ; i < NR_THREAD; i++)
    {
        pthread_create(&tid_2[i],NULL,thread_work,NULL);
        if(ret != 0)
        {
            fprintf(stderr,"create thread failed ,return %d (%s)\n",ret,strerror_r (ret,errbuf,sizeof(errbuf)));
        }
    }
    for(i = 0 ; i < NR_THREAD; i++)
    {
        pthread_join(tid_2[i],NULL);
    }
    printf("join thread End\n");
    exit(0);
}
```

```
        pthread_join(tid[i],NULL);
    }
#endif
    for(i = 0 ; i < NR_THREAD ; i++)
    {
        ret = pthread_create(&tid_2[i],NULL,thread_work,NULL);
        if(ret != 0)
        {
            fprintf(stderr,"create thread failed ,return %d (%s)\n",ret,strerror_r (ret,errbuf,sizeof(errbuf)));
        }
    }
    sleep(1000);
    exit(0);
}
```

根据编译选项NO_JOIN，将程序编译成以下两种情况：

·编译加上-DNO_JOIN：主线不执行pthread_join，主线程通过sleep足够的时间，来确保第一个线程退出以后，再创建第二个线程。

·不加NO_JOIN编译选项：主线程负责连接线程，第一个线程退出以后，再来创建第二个线程。

下面按照编译选项，分别编出pthread_no_join和pthread_has_join两个程序：

```
gcc -o pthread_no_join pthread_join_cmp.c -DNO_JOIN -
```

```
lpthread
gcc -o pthread_has_join pthread_join_cmp.c          -lpthread
```

首先说说pthread_no_join的情况，当创建了第一个线程时：

```
manu@manu-hacks:~/code/me/thread$ ./pthread_no_join
thread-12876 IN
thread-12876 pthread_self return 0x7fe0c842b700
```

从输出可以看到，创建了第一个线程，其线程ID为12876，通过pmap和procfs可以看到系统为该线程分配了8MB的地址空间：

```
manu@manu-hacks:~$ pmap 12875
00007fe0c7c2b000      4K ----- [ anon ]
00007fe0c7c2c000    8192K rw---- [ anon ]
manu@manu-hacks:~$ cat /proc/12875/maps
7fe0c7c2b000-7fe0c7c2c000 ---p 00000000 00:00 07fe0c7c2c000-7fe0c842c000
```

```
rw-p 00000000 00:00 0                                [stack:12876]
```

当线程12876退出，创建新的线程时：

```
thread-12876 EXIT
thread-13391 IN
thread-13391 pthread_self return 0x7fe0c7c2a700
```

此时查看进程的地址空间：

```
00007fe0c742a000      4K ----- [ anon ]
00007fe0c742b000    8192K rw---- [ anon ]
00007fe0c7c2b000      4K ----- [ anon ]
00007fe0c7c2c000    8192K rw---- [ anon ]
7fe0c742a000-7fe0c742b000 ---p 00000000 00:00 0
7fe0c742b000-7fe0c7c2b000 rw-p 00000000 00:00 0                                [stack:13391]
7fe0c7c2b000-7fe0c7c2c000 ---p 00000000 00:00 07fe0c7c2c000-7fe0c842c000
```

```
rw-p 00000000 00:00 0
```

从上面的输出可以看出两点：

1）已经退出的线程，其空间没有被释放，仍然在进程的地址空间之内。

2）新创建的线程，没有复用刚才退出的线程的地址空间。

如果仅仅是情况1的话，尚可以理解，但是1和2同时发生，既不释放，也不复用，这就不能忍了，因为这已经属于内存泄漏了。试想如下场景：FTP Server采用thread per connection的模型，每接受一个连接就创建一个线程为之服务，服务结束后，连接断开，线程退出。但线程退出了，线程栈消耗的空间仍不能释放，不能复用，久而久之，内存耗尽，再也不能创建线程，也无法再提供FTP服务。

之所以不能复用，原因就在于没有对退出的线程执行连接操作。下面来看一下主线程调用pthread_join的情况：

```
manu@manu-hacks:~/code/me/thread$ ./pthread_has_join
join thread Begin
thread-14581 IN
thread-14581 pthread_self return 0x7f726020f700
thread-14581 EXIT
thread-14871 IN
thread-14871 pthread_self return 0x7f726020f700
thread-14871 EXIT
```

两次创建的线程，pthread_t类型的线程ID完全相同，看起来好像前面退出的栈空间被复用了，事实也的确如此：

```
manu@manu-hacks:~$ cat /proc/14580/maps
7f725fa0f000-7f725fa10000 ---p 00000000 00:00 0
7f725fa10000-7f7260210000 rw-p 00000000 00:00 0                                [stack:14581]
```

12581退出后，线程栈被后创建的线程复用了：

```
manu@manu-hacks:~$ cat /proc/14580/maps
7f725fa0f000-7f725fa10000 ---p 00000000 00:00 0
7f725fa10000-7f7260210000 rw-p 00000000 00:00 0                                [stack:14871]
```

通过前面的比较，可以看出执行连接操作的重要性：如果不执行连接操作，线程的资源就不能被释放，也不能被复用，这就造成了资源的泄漏。

当线程组内的其他线程调用pthread_join连接退出线程时，内部会调用__free_tcb函数，该函数会负责释放退出线程的资源。

值得一提的是，纵然调用了pthread_join，也并没有立即调用munmap来释放掉退出线程的栈，它们是被后建的线程复用了，这是NPTL线程库的设计。释放线程资源的时候，NPTL认为进程可能再次创建线程，而频繁地munmap和mmap会影响性能，所以NPTL将该栈缓存起来，放到一个链表之中，如果有新的创建线程的请求，NPTL会首先在栈缓存链表中寻找空间合适的栈，有的话，直接将该栈分配给新创建的线程。

7.6.3 线程的分离

默认情况下，新创建的线程处于可连接（Joinable）的状态，可连接状态的线程退出后，需要对其执行连接操作，否则线程资源无法释放，从而造成资源泄漏。

如果其他线程并不关心线程的返回值，那么连接操作就会变成一种负担：你不需要它，但是你不执行连接操作又会造成资源泄漏。这时候你需要的东西只是：线程退出时，系统自动将线程相关的资源释放掉，无须等待连接。

NPTL提供了pthread_detach函数来将线程设置成已分离（detached）的状态，如果线程处于已分离的状态，那么线程退出时，系统将负责回收线程的资源，如下：

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

可以是线程组内其他线程对目标线程进行分离，也可以是线程自己执行pthread_detach函数，将自身设置成已分离的状态，如下：

```
pthread_detach(pthread_self())
```

线程的状态之中，可连接状态和已分离状态是冲突的，一个线程不能既是可连接的，又是已分离的。因此，如果线程处于已分离的状态，其他线程尝试连接线程时，会返回EINVAL错误。

pthread_detach出错的情况见表7-8所示。

表7-8 pthread_detach的错误码和说明

返 回 值	说 明
ESRCH	传入的线程 ID 不存在，查无此线程
EINVAL	线程不是一个可连接（joinable）的线程，已经处于已分离状态

需要强调的是，不要误解已分离状态的内涵。所谓已分离，并不是指线程失去控制，不归线程组管理，而是指线程退出后，系统会自动释放线程资源。若线程组内的任意线程执行了exit函数，即使是已分离的线程，也仍然会受到影响，一并退出。

将线程设置成已分离状态，并非只有pthread_detach一种方法。另一种方法是在创建线程时，将线程的属性设定为已分离：

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr,int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr,int *detachstate);
```

其中detachstate的可能值如表7-9所示。

表7-9 分离状态的合法值

分离状态的可选值	说 明
PTHREAD_CREATE_JOINABLE	默认情况，表示创建出来的线程会处于可连接的状态
PTHREAD_CREATE_DETACHED	表示创建出来的线程，会处于已分离的状态

有了这个，如果确实不关心线程的返回值，可以在创建线程之初，就指定其分离属性为PTHREAD_CREATE_DETACHED。

7.7 互斥量

7.7.1 为什么需要互斥量

大部分情况下，线程使用的数据都是局部变量，变量的地址在线程栈空间内，这种情况下，变量归属于单个线程，其他线程无法获取到这种变量。

如果所有的变量都是如此，将会省去无数的麻烦。但实际的情况是，很多变量都是多个线程共享的，这样的变量称为共享变量（shared variable）。可以通过数据的共享，完成多个线程之间的交互。

但是多个线程并发地操作共享变量，会带来一些问题。

下面来看一个例子，如图7-12所示。

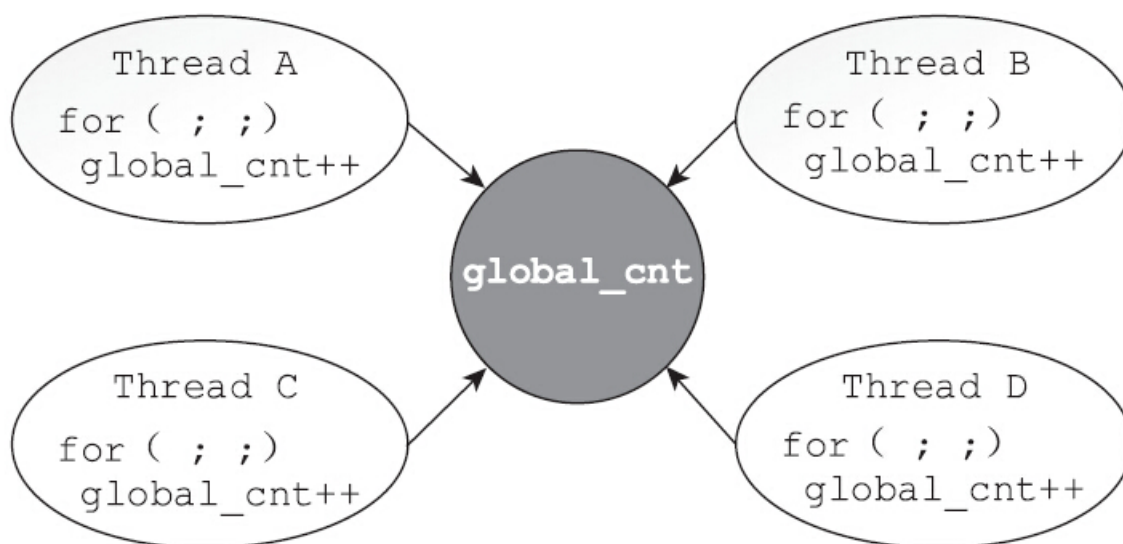


图7-12 多线程操作全局变量

如果存在4个线程，不加任何同步措施，共同操作一个全局变量`global_cnt`，假设每个线程执行1000万次自加操作，那么会发生什么事情呢？4个线程结束的时候，`global_cnt`等于几？

这个问题看起来是小学题目，当然是4000万，但实际结果又如何呢？

```
#define GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#define LOOP_TIMES 10000000
#define NR_THREAD 4
pthread_rwlock_t rwlock;
int global_cnt = 0;
void* thread_work(void* param)
{
    int i;
    pthread_rwlock_rdlock(&rwlock);
```



```

    for(i = 0 ; i < LOOP_TIMES; i++ )
    {
        global_cnt++;
    }
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}
int main(int argc ,char* argv[])
{
    pthread_t tid[NR_THREAD];
    char err_buf[1024];
    int i, ret;
    ret = pthread_rwlock_init(&rwlock,NULL);
    if(ret)
    {
        fprintf(stderr,"init rw lock failed (%s)\n",strerror_r(ret,err_buf, sizeof(err_buf)));
        exit(1);
    }
    pthread_rwlock_wrlock(&rwlock);
    for(i = 0 ; i < NR_THREAD ; i++)
    {
        ret = pthread_create(&tid[i],NULL,thread_work,NULL);
        if(ret != 0)
        {
            fprintf(stderr,"create thread failed ,return %d (%s)\n",
                ret,strerror_r(ret,err_buf,sizeof(err_buf)));
        }
    }
    pthread_rwlock_unlock(&rwlock);
    for(i = 0 ; i < NR_THREAD; i++)
    {
        pthread_join(tid[i],NULL);
    }
    pthread_rwlock_destroy(&rwlock);
    printf("thread_num      : %d\n",NR_THREAD);
    printf("loops per thread : %d\n",LOOP_TIMES);
    printf("expect result    : %d\n",LOOP_TIMES*NR_THREAD);
    printf("actual result    : %d\n",global_cnt);
    exit(0);
}

```

上面的代码中，引入了读写锁，来确保线程位于同一起跑线，同时开始执行自加操作，不受线程创建先后顺序的影响。创建4个线程之前，主线程先占住读写锁的写锁，任一线程创建好了之后，要先申请读锁，申请成功方能执行`global_cnt++`，但是写锁已经被主线程占据，所以无法执行。待4个线程都创建成功后，主线程会释放写锁，从而保证4个线程一起执行。

执行结果又如何呢？来看看：

```

thread num      : 4
loops per thread : 10000000
expect result    : 40000000
actual result    : 11115156

```

结果并不是期待的4000万，而是11115156，一个很奇怪的数字。而且每次执行，最后的结果都不相同。

为什么无法获得正确的结果？

看一下汇编代码，先通过如下指令读取到汇编代码：

```
objdump -d pthread_no_sync > pthread_no_sync.objdump
```

然后在汇编代码中取出`global_cnt++`这部分代码相关的汇编代码，就是如下指令：

```
40098c: 8b 05 1a 07 20 00  mov 0x20071a(%rip),%eax # 6010ac <global_cnt>
400992: 83 c0 01          add $0x1,%eax
400995: 89 05 11 07 20 00  mov %eax,0x200711(%rip) # 6010ac <global_cnt>
```

++操作，并不是一个原子操作（atomic operation），而是对应了如下三条汇编指令。

- Load: 将共享变量global_cnt从内存加载进寄存器，简称L。
- Update: 更新寄存器里面的global_cnt值，执行加1操作，简称U。
- Store: 将新的值，从寄存器写回到共享变量global_cnt的内存地址，简称为S。

将上述情况用伪代码表示，就是如下情况：

```
L操作:

register = global_cnt
U操作:

register = register + 1
S操作:

global_cnt = register
```

以两个线程为例，如果两个线程的执行如图7-13所示，就会引发结果不一致：执行了两次++操作，最终的结果却只加了1。

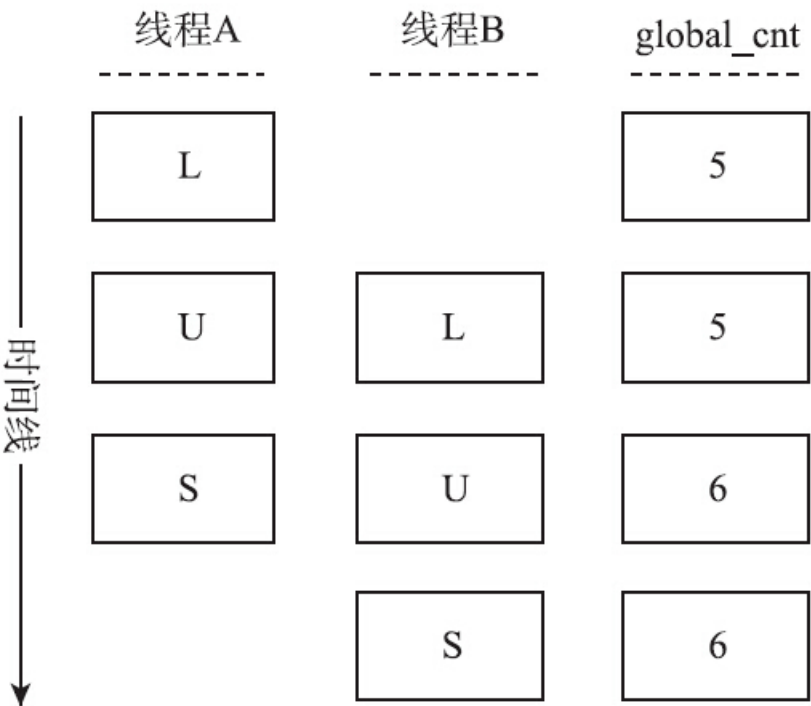


图7-13 多线程操作全局变量结果出错的原因

上面的例子表明，应该避免多个线程同时操作共享变量，对于共享变量的访问，包括读取和写入，都必须被限制为每次只有一个线程来执行。

用更详细的语言来描述下，解决方案需要能够做到以下三点。

- 1) 代码必须要有互斥的行为：当一个线程正在临界区中执行时，不允许其他线程进入该临界区中。
- 2) 如果多个线程同时要求执行临界区的代码，并且当前临界区并没有线程在执行，那么只能允许一个线程进入该临界区。
- 3) 如果线程不在临界区中执行，那么该线程不能阻止其他线程进入临界区。

上面说了这么多，本质其实就是一句话，我们需要一把锁（如图7-14所示）。

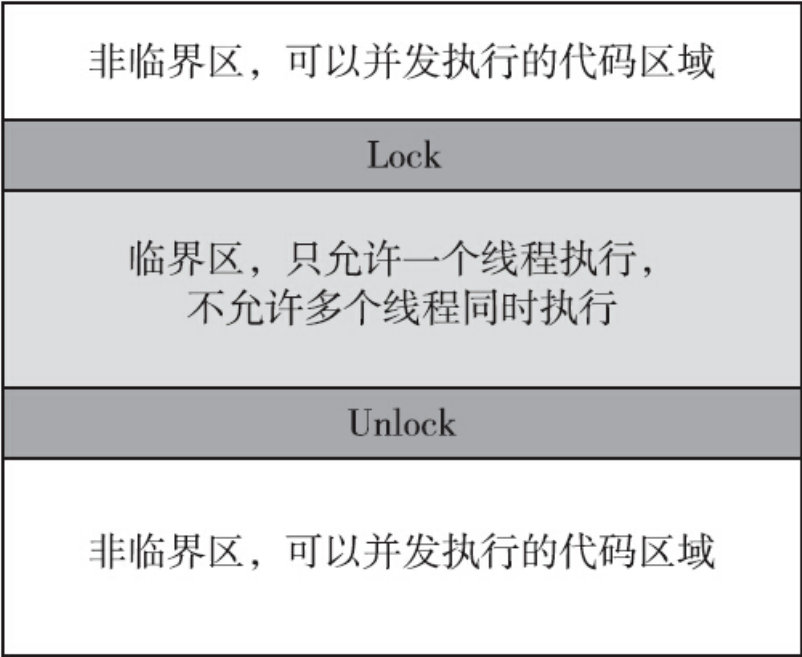


图7-14 用锁来保护临界区

锁是一个很普遍的需求，当然用户可以自行实现锁来保护临界区。但是实现一个正确并且高效的锁非常困难。纵然抛下高效不谈，让用户从零开始实现一个正确的锁也并不容易。正是因为这种需求具有普遍性，所以Linux提供了互斥量。

7.7.2 互斥量的接口

1.互斥量的初始化

互斥量采用的是英文**mutual exclusive**（互相排斥之意）的缩写，即**mutex**。

正确地使用互斥量来保护共享数据，首先要定义和初始化互斥量。POSIX提供了两种初始化互斥量的方法。

第一种方法是将**PTHREAD_MUTEX_INITIALIZER**赋值给定义的互斥量，如下：

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

如果互斥量是动态分配的，或者需要设定互斥量的属性，那么上面静态初始化的方法就不适用了，NPTL提供了另外的函数**pthread_mutex_init()**对互斥量进行动态的初始化：

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

第二个**pthread_mutexattr_t**指针的入参，是用来设定互斥量的属性的。大部分情况下，并不需要设置互斥量的属性，传递**NULL**即可，表示使用互斥量的默认属性。

调用**pthread_mutex_init()**之后，互斥量处于没有加锁的状态。

2.互斥量的销毁

在确定不再需要互斥量的时候，就要销毁它。在销毁之前，有三点需要注意：

- 使用**PTHREAD_MUTEX_INITIALIZER**初始化的互斥量无须销毁。
- 不要销毁一个已加锁的互斥量，或者是真正配合条件变量使用的互斥量。
- 已经销毁的互斥量，要确保后面不会有线程再尝试加锁。

销毁互斥量的接口如下：

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

当互斥量处于已加锁的状态，或者正在和条件变量配合使用，调用**pthread_mutex_destroy**函数会返回**EBUSY**错误码。

3.互斥量的加锁和解锁

POSIX提供了如下接口：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

在调用`pthread_lock()`的时候，可能会遭遇以下几种情况：

- 互斥量处于未锁定的状态，该函数会将互斥量锁定，同时返回成功。
- 发起函数调用时，其他线程已锁定互斥量，或者存在其他线程同时申请互斥量，但没有竞争到互斥量，那么`pthread_lock()`调用会陷入阻塞，等待互斥量解锁。

在等待的过程中，如果互斥量持有线程解锁互斥量，可能会发生如下事件：

- 函数调用线程是唯一等待者，获得互斥量，成功返回。
- 函数调用线程不是唯一等待者，但成功获得互斥量，返回。
- 函数调用线程不是唯一等待者，没能获得互斥量，继续阻塞，等待下一轮。
- 如果在调用`pthread_lock()`线程时，之前已经调用过`pthread_lock()`且已经持有了互斥量，则根据互斥锁的类型，存在以下三种可能。

- `PTHREAD_MUTEX_NORMAL`：这是默认类型的互斥锁，这种情况下会发生死锁，调用线程永久阻塞，线程组的其他线程也无法申请到该互斥量。

- `PTHREAD_MUTEX_ERRORCHECK`：第二次调用`pthread_mutex_lock`函数时返回`EDEADLK`。

- `PTHREAD_MUTEX_RECURSIVE`：这种类型的互斥锁内部维护有引用计数，允许锁的持有者再次调用加锁操作。

有了互斥量，重新运行7.7.1节的程序，将`global_cnt++`改写成：

```
pthread_mutex_lock(&mutex);
global_cnt++;
pthread_mutex_unlock(&mutex);
```

使用互斥量之后，程序获取了正确的执行结果：

```
thread num      : 4
loops per thread : 10000000
expect result    : 40000000
```

actual result : 40000000

7.7.3 临界区的大小

现在，我们已经意识到需要用锁来保护共享变量。不过还有另一个需要注意的事项，即合理地设定临界区的范围。

第一临界区的范围不能太小，如果太小，可能起不到保护的目的。考虑如下场景，如果哈希表中不存在某元素，那么向哈希表中插入某元素，代码如下：

```
if(!htable_contain(hashtable,elem.key))
{
    pthread_mutex_lock(&mutex);
    htable_insert(hashtable,&elem);
    pthread_mutex_unlock(&mutex);
}
```

表面上看，共享变量hashtable得到了保护，在插入时有锁保护，但是结果却不是我们想要的。上面的程序不希望哈希表中有重复的元素，但是其临界区太小，多线程条件下可能达不到预设的效果。

如果时序如图7-15所示，那么就会有重复的元素被插入哈希表中，没有达到最初的目的。究其原因，就是临界区小了，没有将判断部分加入临界区以内。

临界区也不能太大，临界区的代码不能并发，如果临界区太大，就无法充分利用多处理器发挥多线程的优势。对于被互斥量保护的临界区内的代码，一定要好好审视，不要将不相干的（特别是可能陷入阻塞的）代码放入临界区内执行。

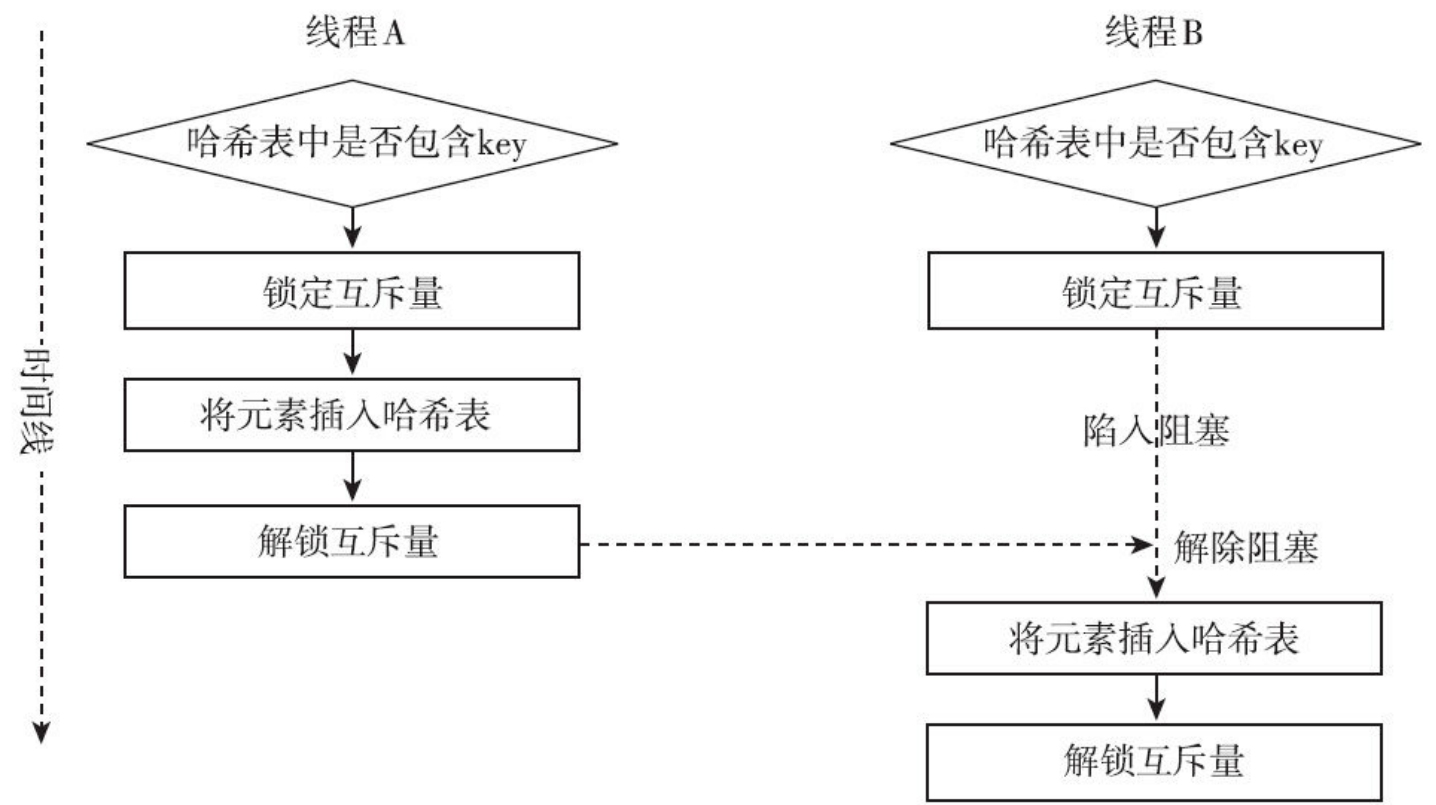


图7-15 临界区太小，未能解决竞争，重复插入了某元素

7.7.4 互斥量的性能

还是以前面的例子为例进行说明，4个线程分别对全局变量累加1000万次，使用互斥量版本的程序和不使用互斥量的版本相比，会消耗更多的时间，如表7-10所示。

表7-10 加锁版本和无锁版本的性能比较

无互斥量的版本			使用互斥量的版本		
real	user	sys	real	user	sys
0.126s	0.402s	0.027s	4.360s	4.617s	11.433s

- 互斥量版本需要消耗更长的时间，其原因有以下三点：
- 1）对互斥量的加锁和解锁操作，本身有一定的开销。
 - 2）临界区的代码不能并发执行。
 - 3）进入临界区的次数过于频繁，线程之间对临界区的争夺太过激烈，若线程竞争互斥量失败，就会陷入阻塞，让出CPU，所以执行上下文切换的次数要远远多于不使用互斥量的版本。

看到这个结果，又有一个疑问涌上心头，互斥量的性能如何？

Linux下，互斥量的实现采用了futex（fast user space mutex）机制。传统的同步手段，进入临界区之前会申请锁，而此时不得不执行系统调用，查看是否存在竞争；当离开临界区释放锁的时候，需要再次执行系统调用，查看是否需要唤醒正在等待锁的进程。但是在竞争并不激烈的情

况下，加锁和解锁的过程中可能会出现以下两种情况：

- 申请锁时，执行系统调用，从用户模式进入内核模式，却发现并无竞争。
 - 释放锁时，执行系统调用，从用户模式进入内核模式，尝试唤醒正在等待锁的进程，却发现并没有进程正在等待锁的释放。
- 考虑到系统调用的开销，这两种情况耗资靡费，却劳而无功。

futex机制的出现有效地解决了这两个问题。futex的全称是fast userspace mutex，中文名为快速用户空间互斥体，它是一种用户态和内核态协同工作的同步机制。glibc使用内核提供的futex系统调用实现了互斥量。

glibc的互斥量实现，含有大量的汇编代码，不易读懂，下面用伪代码来描述下互斥量的加锁和解锁操作：

```
void lock(mutex* lock)
{
    int c;

    if (c = cmpxchg(lock,0,1) != 0)
        // 如果原始值是

    0，则表示处于没加锁的状态，将

    lock改成

    1，直接返回

    // 如果原始值不是

    0，则表示互斥量已被加锁，需要继续执行
```



```
do
{
/* 此处有以下可能性：
```

1) c==2 表示已被加锁，并且有其他正在等待的线程

,应立即调用

futex_wait(2) 原子地检查

lock是否为

1,

如果是，则将

lock改成

2, 然后调用

futex_wait
如果不是，则表示其他线程释放了锁，将

lock改成了

0, 需要执行

while语句争夺锁

```
*/
if (
```

```
c == 2 || cmpxchg(lock, 1, 2) != 0)
{
    //如果执行
```

futex_wait时,

lock已经被改写, 不等于

2, 则当即返回

```
    futex_wait(lock, 2);
} while (
```

```
(c = cmpxchg(lock, 0, 2))!
```

```
= 0)
```

```
;
//表示有线程
```

unlock, 但是不知道解锁后是

1还是

2, 保险起见, 写成

```
2
}
void unlock(mutex* lock)
{
    //atomic_dec的作用是减
```

1并返回原始值

```
if (atomic_dec(lock) != 1)
{
    // 原始值是
```

2, 有线程等待互斥量, 才会进入

```
// 如果原始值是
```

1, 则表示没有线程等待, 没必要

```
futex_wake
{
    lock = 0;
    futex_wake(lock, 1);
}
}
```

上面的cmpxchg和atomic_dec都是原子操作。

·cmpxchg (lock, a, b): 表示如果lock的值等于a, 那么将lock改为b, 并将原始值返回, 否则直接将原始值返回。

·atomic_dec (lock): 表示将lock的值减去1, 并且返回原始值。

glibc的互斥量中维护了一个值lock, 该值有以下三种情况。

·0: 表示互斥量并未上锁。

·1: 表示互斥量已经上锁, 但是并没有线程正在等待该锁。

·2: 表示互斥量已经上锁, 并且有线程正在等待该锁。

加锁时, 如果发现该值是0, 那么直接将该值改为1, 无须执行任何系统调用, 因为并没有线程持有该锁, 无须等待;

解锁时, 如果发现该值是1, 直接将该值改成0, 无须执行任何系统调用, 因为并没有线程正在等待该锁, 无须唤醒。

当然, 在这两种情况下, 比较和修改操作 (Compare And Swap) 必须是原子操作, 否则会出现问题。如果无竞争, 可以看出, 互斥量的加锁和解锁非常轻量级。

用一个简单的实验也可以证明, 无竞争条件下, 加锁解锁的操作是很轻量级的。下面用一个循环执行加锁和解锁操作1000万次, 统计下加锁解锁一次消耗的平均时间, 即:

```
clock_gettime(CLOCK_MONOTONIC, &start);
for (int i = 0; i < TIMES; ++i) {
    pthread_mutex_lock(&lock);
    pthread_mutex_unlock(&lock);
}
clock_gettime(CLOCK_MONOTONIC, &end);
```

在笔者用的2.13GHz i3处理器的Ubuntu上, 加锁解锁一次, 平均消耗24纳秒左右, 证明了在无竞争的条件下, 互斥量的加锁和解锁操作的确是十分轻量级的。

接下来考虑存在竞争的情况, 这时候, 就需要内核来参与了。

内核提供了futex_wait和futex_wake两个操作 (futex系统调用支持的两个命令):

```
int futex_wait(int *uaddr, int val);
```

```
int futex_wake(int *uaddr, int n);
```

`futex_wait`是用来协助加锁操作的。线程调用`pthread_mutex_lock`，如果发现锁的值不是0，就会调用`futex_wait`，告知内核，线程须要等待在uaddr对应的锁上，请将线程挂起。内核会建立与uaddr地址对应的等待队列。

为什么需要内核维护等待队列？因为一旦互斥量的持有者线程释放了互斥量，就需要及时通知那些等待在该互斥量上的线程。如果没有等待队列，内核将无法通知到那些正陷入阻塞的线程。

如果整个系统有很多这种互斥量，是不是需要为每个uaddr地址建立一个等待队列呢？事实上不需要。理论上讲，`futex`只需要在内核之中维护一个队列就够了，当线程释放互斥量时，可能会调用`futex_wake`，此时会将uaddr传进来，内核会去遍历该队列，查找等待在该uaddr地址上的线程，并将相应的线程唤醒。

但是只有一个队列的话查找效率有点低，作为优化，内核实现了多个队列。插入等待队列时，会先计算hash值，然后根据hash插入到对应的链表之中，如图7-16所示。

值得一提的是，`futex_wait`操作需要的val入参，乍看之下好像没什么用处。事实上并非如此。从用户程序判断锁的值，到调用`futex_wait`操作是有时间窗口的，在这个时间窗口之内，有可能发生线程解锁的操作，从而可能无须等待。因此`futex_wait`操作会检查uaddr对应的锁的值是否等于val的值，只有在等于val的情况下，内核才会让线程等待在对应的队列上，否则会立刻返回，让用户程序再次申请锁。

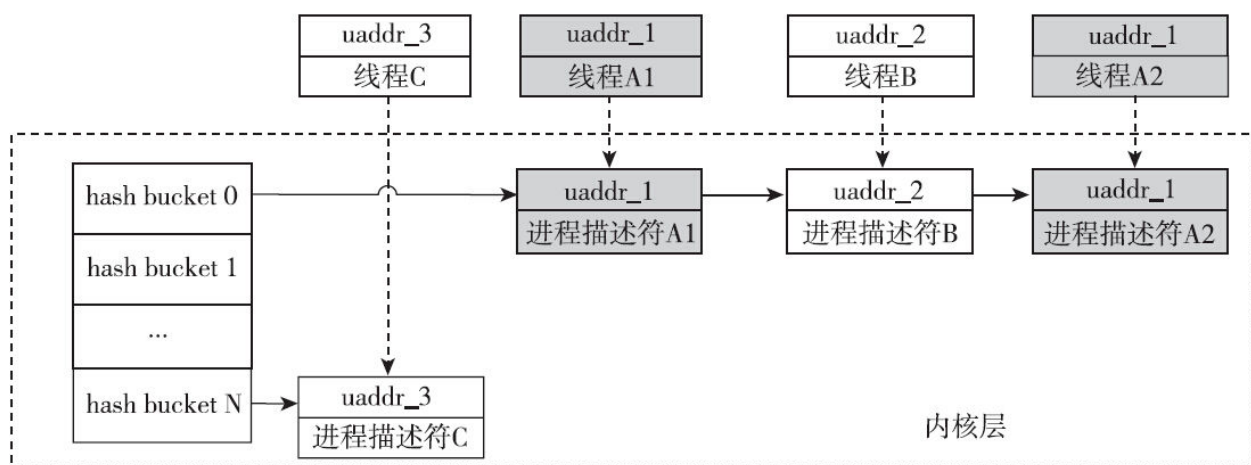


图7-16 futex机制中内核的等待队列

`futex_wake`操作是用来实现解锁操作的。`glibc`就是使用该操作来实现互斥量的解锁函数`pthread_mutex_unlock`的。当线程执行完临界区代码，解锁时，内核需要通知那些正在等待该锁的线程。这时候就需要发挥`futex_wake`操作的作用了。`futex_wake`的第二个参数n，对于互斥量而言，该值总是1，表示唤醒1个线程。当然，也可以唤醒所有正在等待该锁的线程，但是这样做并无好处，因为被唤醒的多个线程会再次竞争，却只能有一个线程抢到锁，这时其他线程不得不再次睡去，徒增了很多开销。

使用`strace`跟踪系统调用的时候，看不到`futex_wait`和`futex_wake`两个系统调用，看到的是`futex`系统调用，如下。

```
#include <linux/futex.h>
#include <sys/time.h>
int futex(int *uaddr, int op, int val,
const struct timespec *timeout,int *uaddr2, int val3);
```

该系统调用是一个综合的系统调用，根据第二个参数op来决定具体的行为。当op为FUTEX_WAIT时，对应的是前面讨论的`futex_wait`操作，当op为FUTEX_WAKE时，对应的是前面讨论的`futex_wake`操作。

细心的话，可以发现，互斥量加锁和解锁时，调用futex的op参数并非FUTEX_WAIT和FUTEX_WAKE，而是FUTEX_WAIT_PRIVATE和FUTEX_WAKE_PRIVATE，这是为了改进futex的性能而进行的优化。因为futex也可以用在不同的进程之间，加上后缀_PRIVATE是为了明确告知内核，互斥的行为是用在线程之间的。

从上面的角度分析，当存在竞争时，如果线程申请不到互斥量，就会让出CPU，系统会发生上下文切换。在线程个数众多，临界区竞争异常激烈的情况下，上下文切换会是一笔不小的开销。

如果临界区非常小，线程之间对临界区的竞争并不激烈，只会偶尔发生，这种情况下，忙-等待的策略要优于互斥量的“让出CPU，陷入阻塞，等待唤醒”的策略。采用忙-等待策略的锁为自旋锁。

关于futex的原理，Ulrich Drepper《Futexes Are Tricky》^[1]一文就是非常好的参考文献。

^[1] Ulrich Drepper的《Futexes Are Tricky》，详见<http://www.akkadia.org/drepper/futex.pdf>。

7.7.5 互斥锁的公平性

互斥锁是公平的吗？

首先要定义什么是公平（fairness）。对于锁而言，如果A在B之前调用lock（）方法，那么A应该先于B获得锁，进入临界区。多处理器条件下，很难确定是哪个线程率先调用的lock（）方法。纵然能判定是哪个线程率先调用的lock（）方法，要实现指令级的公平也是很难的。常见的判断锁公平性的方法是，将锁的实现代码分成如下两个部分：

- 门廊区

- 等待区

门廊区必须在有限的操作内完成，等待区则可能有无穷的步骤，它们会陷入未知结束时间的等待中。

如果锁能满足以下条件，就称锁是先来先服务（FCFS）的：

如果线程A门廊区的结束在线程B门廊区的开始之前，那么线程A一定不会被线程B赶超。

互斥量也有门廊区和等待区，就像7.7.4节分析的，如果没有竞争，线程执行几个指令就加锁成功，顺利返回了。在这种情况下，互斥量在门廊区就解决了所有的需要。但是如果有竞争，互斥锁在门廊区判断出存在竞争，线程取不到锁，就不得不执行futex_wait，让内核将其挂起，并记录在等待队列上。需要等待多久？不知道。

从表面上看，内核会将等待互斥量的线程放入队列，每来一个等待线程，就把线程记录在队列的尾部，当互斥量的持有线程解锁时，内核只会唤醒一个线程，而唤醒的正是队列中等待该互斥量的第一个等待者。队列的先入先出（FIFO），看起来已经保证了互斥量的公平性。但是，这样就能确保公平吗？

答案是否定的，互斥锁并没有做到先来先服务。

根据7.7.4节的伪代码可知，当互斥量的lock的值是2，或者尝试调用CAS操作将lock从1改成2并且成功时，线程会调用futex_wait陷入阻塞。值得一提的是，CAS操作在尝试将1改成2时，也可能存在竞争，比如其他线程有解锁操作，lock值已经被改成了0，而这时候恰好存在另外一个线程刚刚调用加锁操作，这时就会发生门廊区的争夺，对于这种情况不做详细分析。假设加锁调用了futex_wait，内核将线程挂起在等待队列上，从那时起，线程就进入了漫长的等待区。

如果互斥量的持有线程解锁，会首先将互斥量的lock值设置成0，然后唤醒内核等待队列中等待在该地址上的第一个线程。看起来比较公平，但是问题就出在此处，被唤醒的线程并不是自动就持有了互斥锁，反而须要执行while（）中包裹的cmpxchg操作，再次竞争互斥量。如果竞争失败，则被另外一个初来乍到的线程将0改成了1，那么线程刚刚醒来就不得不再次执行futex_wait，再次沉睡。这次竞争失败的代价是巨大的，因为futex_wait操作会将线程挂载到等待队列的队尾。

由上面的分析可以得出如下结论：

- 线程可能多次调用futex_wait进入等待区，在线程被futex_wait唤醒后，并不会自动拥有互斥量，而是再次进入门廊区，和其他线程争夺锁。

- 在已经有很多线程处于内核等待队列的情况下，新来的加锁请求可能会后发先至，率先获得锁。

- futex_wait唤醒的线程如果没有竞争到锁，那么会再次调用futex_wait函数，陷入睡眠，不过内核会将其放入等待队列的队尾，这种行为加剧了不公平性。

所以，综合上面的讨论，互斥量不是一个公平的锁，没有做到先来先服务。关于futex的早期论文《Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux》，已经指出了这个问题。futex_up_fair系统调用尝试解决这个不公平的问题，但是最终没有进入内核主线。

为什么开发者并不在意这种不公平性？因为要实现这种公平性会牺牲性能，而这种牺牲并无必要。绝大多数情况下，由于调度的原因，用户根本无法判断哪个线程会优先调用加锁操作，那么内核或glibc维持这种先来先服务（FCFS）就变得毫无意义。如果可以在不牺牲性能的情况下做到公平，自然最好，但是实际情况并非如此。实现这种公平，对性能的伤害很大。就像Ulrich Drepple在Thread starvation with mutex的回复中所说的：

```
Is there a reason why NPTL does not use this "fair" method?  
It's slow and unnecessary.
```

综上所述，结论如下：内核维护等待队列，互斥量实现了大体上的公平；由于等待线程被唤醒后，并不自动持有互斥量，需要和刚进入门廊区的线程竞争，所以互斥量并没有做到先来先服务。

7.7.6 互斥锁的类型

前面讨论的都是默认类型的互斥锁，除默认类型外，互斥锁还有几个变种，它们的行为模式和默认互斥锁有一定的差异。

互斥量有以下4种类型：

- PTHREAD_MUTEX_TIMED_NP
- PTHREAD_MUTEX_RECURSIVE
- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_ADAPTIVE_NP

glibc提供了接口来查询和设置互斥锁的类型：

```
#include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr,int type);
```

可以仿照如下代码来设置互斥量的类型：

/*忽略了出错判断，真实代码中需要判断

```
error*/
pthread_mutex_t mtx;
pthread_mutexattr_t mtxAttr;
pthread_mutexattr_init(&mtxAttr);
pthread_mutexattr_settype(&mtxAttr,PTHREAD_MUTEX_ADAPTIVE_NP);
pthread_mutex_init(&mtx,&mtxAttr);
```

其中manual给出了4种类型，但并非前面提到的这4种类型，略有差异，差异在于：manual中存在PTHREAD_MUTEX_DEFAULT类型，而少了一个PTHREAD_MUTEX_ADAPTIVE_NP类型。manual中给出的是标准unix 98定义的4种类型。

对于NPTL的实现，具体如下：

```
PTHREAD_MUTEX_NORMAL = PTHREAD_MUTEX_TIMED_NP,
PTHREAD_MUTEX_DEFAULT = PTHREAD_MUTEX_NORMAL;
```

所以，glibc的实现比标准的Unix 98多了一个PTHREAD_MUTEX_ADAPTIVE_NP类型，下面来分别

介绍这几个互斥量的特点。

·**PTHREAD_MUTEX_NORMAL**：最普通的一种互斥锁。前文讨论的就是这种类型的锁。它不具备死锁检测功能，如线程对自己锁定的互斥量再次加锁，则会发生死锁。

·**PTHREAD_MUTEX_RECURSIVE_NP**：支持递归的一种互斥锁，该互斥量的内部维护有互斥锁的所有者和一个锁计数器。当线程第一次取到互斥锁时，会将锁计数器置1，后续同一个线程再次执行加锁操作时，会递增该锁计数器的值。解锁则递减该锁计数器的值，直到降至0，才会真正释放该互斥量，此时其他线程才能获取到该互斥量。解锁时，如果互斥量的所有者不是调用解锁的线程，则会返回EPERM。

·**PTHREAD_MUTEX_ERRORCHECK_NP**：支持死锁检测的互斥锁。互斥量的内部会记录互斥锁的当前所有者的线程ID（调度域的线程ID）。如果互斥量的持有线程再次调用加锁操作，则会返回EDEADLK。解锁时，如果发现调用解锁操作的线程并不是互斥锁的持有者，则会返回EPERM。

终于轮到**PTHREAD_MUTEX_ADAPTIVE_NP**这种类型了。这种类型堪称互斥锁中的战斗机，特点就是——快，libc的文档里面直接将其称为**fast mutex**。那么它和普通的互斥量相比有何差异，它是如何快速实现的呢？

所有锁的实现都会面临一个相同的问题：加锁时竞争失败了该怎么办？普通互斥量的做法是立刻调用**futex_wait**，陷入阻塞，让出CPU，安静地等待内核将其唤醒。在临界区非常小且很少发生竞争的情况下，这种策略并不算好，因为如果该线程肯自旋，很可能只需要极短的时间，它就能等到锁的持有线程解锁，继续执行。而调用**futex_wait**，执行系统调用和上下文切换的开销可能远大于自旋。

出于这种考虑，glibc引入了线程自旋锁。自旋锁采用了和互斥量完全不同的策略，自旋锁加锁失败，并不会让出CPU，而是不停地尝试加锁，直到成功为止。这种机制在临界区非常小且对临界区的争夺并不激烈的场景下，效果非常好，如下。

```
#include <pthread.h>
int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

自旋锁的效果好，但是副作用也大，如果使用不当，自旋锁的持有者迟迟无法释放锁，那么，自旋接近于死循环，会消耗大量的CPU资源，造成CPU使用率飙升。因此，使用自旋锁时，一定要确保临界区尽可能地小，不要有系统调用，不要调用**sleep**。使用**strcpy/memcpy**等函数也需要谨慎判断操作内存的大小，以及是否会引起缺页中断。

自旋锁副作用大，而互斥量在某些情况下效率可能不够高，有没有一种方法能够结合两种方法的长处呢？

答案是肯定的。这就是PTHREAD_MUTEX_ADAPTIVE_NP类型的互斥量，也被称为自适应锁。大多数操作系统（Solaris、Mac OS X、FreeBSD）都有类似的接口，如果竞争锁失败，首先与自旋锁一样，持续尝试获取，但过了一定时间仍然不能申请到锁，就放弃尝试，让出CPU并等待。

PTHREAD_MUTEX_ADAPTIVE_NP类型的互斥量，采用的就是这种机制，如下：

```
if (LLL_MUTEX_TRYLOCK (mutex) != 0)
{
    int cnt = 0;
    int max_cnt = MIN (MAX_ADAPTIVE_COUNT,
                      mutex->__data.__spins * 2 + 10);
    do
    {
        if (cnt++ >= max_cnt)
        {
            /*自旋也没有等到锁，只能睡去*/

            /*
                LLL_MUTEX_LOCK (mutex);
                break;
            */
        }
#ifdef BUSY_WAIT_NOP
        BUSY_WAIT_NOP;
#endif
    }
    while (LLL_MUTEX_TRYLOCK (mutex) != 0);
    mutex->__data.__spins += (cnt - mutex->__data.__spins) / 8;
}
```

到底等待多长时间才合适呢？这种互斥量定义了一个名为__spins的变量，该值和MAX_ADAPTIVE_COUNT共同决定自旋多久。该类型之所以叫自适应（ADAPTIVE），是因为带有反馈机制，它会根据实际情况，智能地调整__spins的值。

```
mutex->__data.__spins += (cnt - mutex->__data.__spins) / 8;
```

当然自旋不是无止境的向上增长时，MAX_ADAPTIVE_COUNT决定了上限，即调用BUSY_WAIT_NOP的最大次数：

```
# define MAX_ADAPTIVE_COUNT 100
```

对于7.7.1节中对global_cnt自加1000万次的程序，如果把for循环体内的锁换成自适应互斥锁，会比普通的互斥量更快吗？答案是否定的，在这种时时刻刻要加锁和解锁的激烈竞争下，让其他线程睡去，利用上下文切换的时间间隔，让一个线程飞快地自加，执行时间反而是最短的。

但是，真实场景下临界区的争夺不可能激烈到这种程度，如果竞争真的激烈到这种程度，那首先需要反省的是设计问题。在临界区非常小，偶尔发生竞争的情况下，自适应互斥锁的性能要优于普通的互斥锁。

7.7.7 死锁和活锁

对于互斥量而言，可能引起的最大问题就是死锁（dead lock）了。最简单、最好构造的死锁就是图7-17所示的这种场景了。

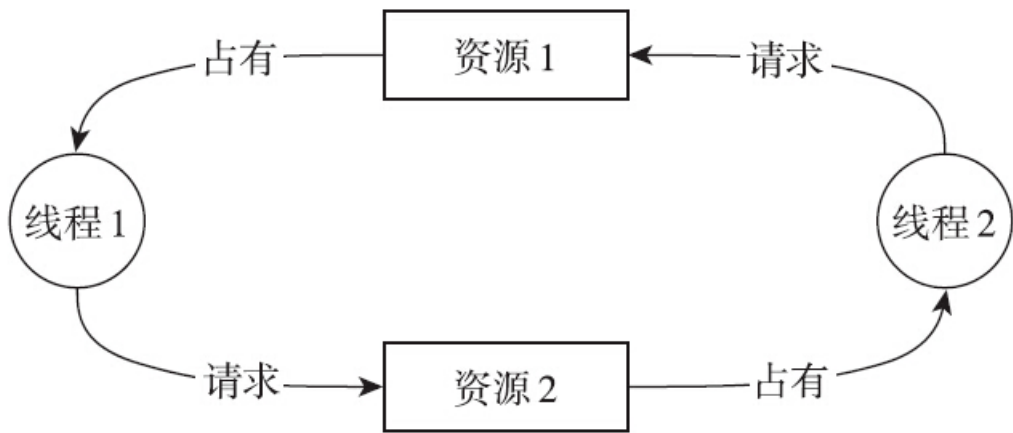


图7-17 死锁的产生（简单场景）

线程1已经成功拿到了互斥量1，正在申请互斥量2，而同时在另一个CPU上，线程2已经拿到了互斥量2，正在申请互斥量1。彼此占有对方正在申请的互斥量，结局就是谁也没办法拿到想要的互斥量，于是死锁就发生了。

上面的例子比较简单，但实际工程中死锁可能会发生在复杂的函数调用之中。可以想象随着程序复杂度的增加，很多死锁并不像上面的例子那样一目了然，如图7-18所示。

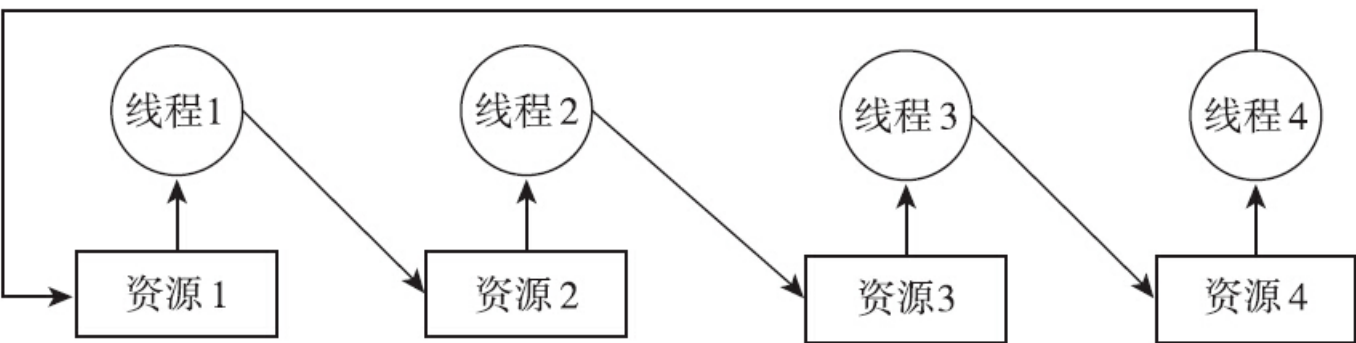


图7-18 死锁的产生（复杂场景）

在多线程程序中，如果存在多个互斥量，一定要小心防范死锁的形成。

存在多个互斥量的情况下，避免死锁最简单的方法就是总是按照一定的先后顺序申请这些互斥量。还是以刚才的例子为例，如果每个线程都按照先申请互斥量1，再申请互斥量2的顺序执行，死锁就不会发生。有些互斥量有明显的层级关系，但是也有一些互斥量原本就没有特定的层级关系，不过没有关系，可以人为干预，让所有的线程必须遵循同样的顺序来申请互斥量。

另一种方法是尝试一下，如果取不到锁就返回。Linux提供了如下接口来表达这种思想：

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t?*restrict mutex, const struct timespec *restrict abs_timeout);
```

这两个函数反应了这种尝试一下，不行就算了的的思想。

对于pthread_mutex_trylock（）接口，如果互斥量已然被锁定，那么当即返回EBUSY错误，而不像pthread_mutex_lock（）接口一样陷入阻塞。

对于pthread_mutex_timedlock（）接口，提供了一个时间参数abs_timeout，如果申请互斥量的时候，互斥量已被锁定，那么等待；如果到了abs_timeout指定的时间，仍然没有申请到互斥量，那么返回ETIMEDOUT错误。

除此以外，这两个接口的表现与pthread_mutex_lock是一致的。在实际的应用中，这两个接口使用的频率远低于pthread_mutex_lock函数。

trylock不行就回退的思想有可能会引发活锁（live lock）。生活中也经常遇到两个人迎面走来，双方都想给对方让路，但是让的方向却不协调，反而互相堵住的情况（如图7-19所示）。活锁现象与这种场景有点类似。



图7-19 让路总让到一起，变成堵路

考虑下面两个线程，线程1首先申请锁mutex_a后，之后尝试申请mutex_b，失败以后，释放mutex_a进入下一轮循环，同时线程2会因为尝试申请mutex_a失败，而释放mutex_b，如果两个线程恰好一直保持这种节奏，就可能在很长的时间内两者都一次次地擦肩而过。当然这毕竟不是死锁，终究会有一个线程同时持有两把锁而结束这种情况。尽管如此，活锁的确会降低性能。这种情况的示例代码如下：

```
//线程
```

```
void func1()
{
    int done = 0;
    while(!done)
    {
        pthread_mutex_lock(&mutex_a);
        if (pthread_mutex_trylock(&mutex_b))
        {
            counter++;
            pthread_mutex_unlock(&mutex_b);
            pthread_mutex_unlock(&mutex_a);
            done = 1;
        }
        else
        {
            pthread_mutex_unlock(&mutex_a);
        }
    }
}
// 线程
```

```
2
void func2()
{
    int done = 0;
    while(!done)
    {
        pthread_mutex_lock (&mutex_b);
        if (pthread_mutex_trylock (&mutex_a))
        {
            counter++;
            pthread_mutex_unlock (&mutex_a);
            pthread_mutex_unlock (&mutex_b);
            done = 1;
        }
        else
        {
            pthread_mutex_unlock (&mutex_b);
        }
    }
}
```

7.8 读写锁

很多时候，对共享变量的访问有以下特点：大多数情况下线程只是读取共享变量的值，并不修改，只有极少数情况下，线程才会真正地修改共享变量的值。

对于这种情况，读请求之间是无需同步的，它们之间的并发访问是安全的。然而写请求必须锁住读请求和其他写请求。

这种情况在实际中是存在的，比如配置项。大多数时间内，配置是不会发生变化的，偶尔会出现修改配置的情况。如果使用互斥量，完全阻止读请求并发，则会造成性能的损失。

出于这种考虑，POSIX引入了读写锁。

读写锁比较简单，从表7-11可以看出，对于这种情况，读写锁做了优化，允许大家一起读。

表7-11 读写锁的行为

当前锁状态	读 锁 请 求	写 锁 请 求
无锁	OK	OK
读锁	OK	阻塞
写锁	阻塞	阻塞

7.8.1 读写锁的接口

1.创建和销毁读写锁

NTPL提供了pthread_rwlock_t类型来表示读写锁。和互斥量一样，它也提供了两种初始化的方法：

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

对于静态变量，可以采用PTHREAD_RWLOCK_INITIALIZER赋值的方式初始化，对于动态分配的读写锁，或者非默认属性的读写锁，需要用pthread_rwlock_init函数进行初始化。如果第二个属性的参数为NULL，那么采用默认属性。

读写锁的默认属性如表7-12所示。

表7-12 读写锁的默认属性

属 性	值	说 明
竞争范围	PTHREAD_PROCESS_PRIVATE	进程内部竞争读写锁
策略	PTHREAD_RWLOCK_PREFER_READER_NP	读者优先

所谓读者优先的策略，是指当前锁的状态是读锁，如果线程申请读锁，此时纵然有写锁在等待队列上，仍然允许申请者获得读锁，而不是被写锁阻塞。后面会详细讨论读者优先和写者优先对读写锁的影响。

对于调用pthread_rwlock_init初始化的读写锁，在不需要读写锁的时候，需要调用pthread_rwlock_destroy销毁，如下：

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

2.读写锁的加锁和解锁

读写锁又称共享-独占锁，有共享，也有独占。

下面是三个读锁上锁的接口：

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,const struct timespec *abstime);
```

而下面三个是写锁上锁的接口：

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,const struct timespec *abstime);
```

读锁用于共享模式。如果当前读写锁已经被某线程以读模式占有了，那么其他线程调用pthread_rwlock_rdlock会立刻获得读锁；如果当前读写锁已经被某线程以写模式占有了，那么调用pthread_rwlock_rdlock会陷入阻塞。

写锁用的是独占模式。如果当前读写锁被某线程以写模式占有，则不允许任何读锁请求通过，也不允许任何写锁请求通过，读锁请求和写锁请求都要陷入阻塞，直到线程释放写锁。

无论是读锁还是写锁，锁的释放都是一个接口：

```
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

无论是读锁还是写锁，都提供了trylock的功能，当不能获得读锁或写锁时，调用线程不会阻塞，而会立即返回，错误码是EBUSY。

无论是读锁还是写锁都提供了限时等待，如果不能获取读写锁，则会陷入阻塞，最多等待到abstime，如果仍然无法获得锁，则返回，错误

码是ETIMEOUT。

从表面上看，读写锁介绍到此处就可以打完收工了，其实不然，读写锁是两种类型的锁，当它们都存在时，它们之间的竞争关系如何？如果同时到来一大拨读锁请求和写锁请求，它们之间的响应又有什么特点？事实上，这些是由读写锁的策略决定的。

7.8.2 读写锁的竞争策略

读写锁的属性是pthread_rwlockattr_t类型，属性中有两个部分：lockkind和pshared。本节只讲lockkind。

所谓lockkind，表示读写锁表现出什么样的行为艺术。对于读写锁，目前有两种策略，一是读者优先，一是写者优先。

glibc引入了如下接口来查询和改变读写锁的类型：

```
int pthread_rwlockattr_getkind_np(const pthread_rwlockattr_t * attr, int * pref);
int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t * attr, int * pref);
```

其中，读写锁类型的可能值有如下几种：

```
enum
{
    PTHREAD_RWLOCK_PREFER_READER_NP, //读者优先

    PTHREAD_RWLOCK_PREFER_WRITER_NP, //很贱人，但是也是读者优先

    PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP, //写者优先

    PTHREAD_RWLOCK_DEFAULT_NP = PTHREAD_RWLOCK_PREFER_READER_NP
};
```

前两个都是读者优先的策略，尤其要注意其中的第二个，名字取得很“变态”，名为PREFER_WRITE却干着“挂羊头卖狗肉”的勾当。只有第三个是写者优先的策略。从pthread_rwlock_init函数中可以看出端倪：

```
rwlock->_data._flags
= iattr->lockkind == PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP;
```

可以看到，只有PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP是写者优先，其他一律都是读者优先。读写锁的默认行为是读者优先。

那么，什么是读者优先呢？

如果当前锁的状态是读锁，并存在写锁请求被阻塞，那么在写锁后面到来的读锁请求该如何处理就成了问题的关键。

如果在写锁请求后面到来的读锁请求不被写锁请求阻塞，就可以立即响应，写锁的下场可能会比较悲惨。如果读锁请求前赴后继源源不断地到来，只要有一个读锁没完成，写锁就没份。这就是所谓的读者优先。

从图7-20可以看出，这种策略是不公平的，极端情况下，写请求很可能被饿死。这就是多线程中的饥饿（Starvation）现象，即某些线程总是得不到锁资源。

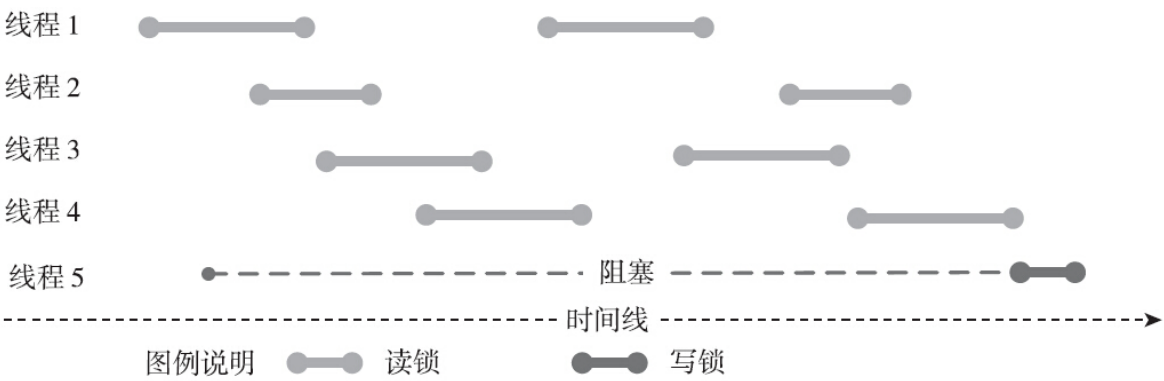


图7-20 较早到的写锁请求被饿死

晚于写锁请求到来的读锁请求不排队乱加塞的行为引起了写锁申请者的强烈不满：凭啥仅仅因为当前是读锁，比我晚来的读锁申请者就不用排队，直接响应？鉴于此，glibc又实现了写者优先的策略。

所谓写者优先是指，如果当前是读锁，有很多线程在共享读锁，这是允许的，但是一旦线程申请写锁，在写锁请求后面到来的读锁请求就会统统被阻塞，不能先于写请求拿到锁。

glibc是如何做到这点的？它引入了表7-13中的变量。

表7-13 读写锁实现中的变量及含义

变 量	说 明
__lock	管理读写锁全局竞争的锁，无论是读锁写锁还是解锁，都会执行互斥
__writer	写锁持有者的线程 ID，如果为 0 则表示当前无线程持有写锁
__nr_readers	读锁持有线程的个数
__nr_readers_queued	读锁的排队等待线程的个数
__nr_writers_queued	写锁的排队等待线程的个数

无论是申请读锁还是申请写锁，还是解锁，都至少会做一次全局互斥锁（对应__lock）的加锁和解锁，若不考虑阻塞，单单考虑操作本身的开销，读写锁的加解锁开销是互斥锁的两倍。当然，函数结束前或进入阻塞之前，会将全局的互斥锁释放。下面的讨论先暂时忽略该全局的互斥锁。

- 对于读锁请求而言，如果：
- 无线程持有写锁，即__writer==0。
 - 采用的是读者优先策略或没有写锁等待者（__nr_writers_queued=0）。

当满足这两个条件时，读锁请求都可以立刻获得读锁，返回之前执行__nr_readers++，表示多了一个线程占有读锁。

不满足的话，则执行__nr_readers_queued++，表示增加一个读锁等待者，然后调用futex，陷入阻塞。醒来之后，会先执行__nr_readers_queued--，然后再次判断是否同时满足条件1和2。

- 对于写请求而言，如果：
- 无线程持有写锁，即__writer==0。
 - 没有线程持有读锁，即__nr_readers==0。

只要满足上述条件，就会立刻拿到写锁，将__writer置为线程的ID（调度域）。

如果不满足，那么执行__nr_writers_queued++，表示增加一个写锁等待者线程，然后执行futex陷入等待。醒来后，限制性__nr_writers_queued--，然后重新判断条件1和2。

对于解锁而言，如果当前锁是写锁，则执行如下操作：

- 1) 执行__writer=0，表示释放写锁。
- 2) 根据__nr_writers_queued判断有没有写锁等待者，如果有，则唤醒一个写锁等待者。

如果没有写锁等待者，则判断有没有读锁等待者；如果有，则将所有读锁等待者一起唤醒。

如果当前锁是读锁，则执行如下操作：

- 1) 执行__nr_readers--，表示读锁占有者少了一个。

2) 判断__nr_readers是否等于0，是的话则表示自己是最后一个读锁占有者，需要唤醒写锁等待者或读锁等待者：

·根据__nr_writers_queued判断是否存在写锁等待者，若有，则唤醒一个写锁等待线程。

·如果没有写锁等待者，判断是否存在读锁等待者，若有，则唤醒所有的读锁等待者。

从上面的流程可以看出，写者优先也存在自私的倾向，因为写锁解锁的时候，首先会去查找有没有阻塞的写锁请求，如果有，先唤醒写锁请求线程。因此如果当前读写锁状态是写锁，同时到来很多写请求和读请求，那它将总是优先处理写请求。如果写锁请求源源不断地到来，那它一样会将读锁请求饿死。

通过上面的分析可以看到，如果存在大量的读写请求，竞争非常激烈的条件下，读写锁存在很大的惯性，如果当前锁的状态是读锁状态，在读者优先的策略下，几乎总是读锁请求先得到响应，写锁被阻塞，因此会出现写请求被饿死的情况。解决的方法是设定成写者优先。如果当前锁的状态是写锁，而写锁也源源不断地到来，这时候，读请求就会被饿死。

下面是一个读写锁的程序，用来验证这种惯性：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N_THREAD 100
static int share_cnt = 0;
static pthread_rwlock_t rwlock ;
void *reader(void *param)
{
    int i = (int) param;
    while(1)
    {
        pthread_rwlock_rdlock(&rwlock) ;
        fprintf(stderr,"reader-%d: the share_cnt = %d\n",i,share_cnt);
        pthread_rwlock_unlock(&rwlock);
    }
    return NULL;
}
void *writer(void *param)
{
    int i = (int) param;
    while(1)
    {
        pthread_rwlock_wrlock(&rwlock) ;
        share_cnt++;
        fprintf(stderr,"writer-%d: the share_cnt = %d\n",i,share_cnt);
        pthread_rwlock_unlock(&rwlock);
        // sleep(1);
    }
    return NULL;
}
int main()
{
    pthread_t tid[N_THREAD] ;
    pthread_rwlockattr_t rwlock_attr ;
    pthread_rwlockattr_init(&rwlock_attr);
#ifdef WRITE_FIRST
    pthread_rwlockattr_setkind_np(&rwlock_attr,PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
#endif
    pthread_rwlock_init(&rwlock,&rwlock_attr);
    int i = 0;
    int ret = 0;
    pthread_rwlock_rdlock(&rwlock);
    for(i = 0;i < N_THREAD; i++)
    {
        if(i%2 == 0)
        {
            ret = pthread_create(&tid[i],NULL,reader, (void*)i);
        }
        else
        {
            ret = pthread_create(&tid[i],NULL,writer, (void*)i);
        }
        if(ret != 0)
        {
            fprintf(stderr,"create thread %d failed \n",i);
            break;
        }
    }
    pthread_rwlock_unlock(&rwlock);
    while(i-- >0)
    {
        pthread_join(tid[i],NULL);
    }
    pthread_rwlockattr_destroy(&rwlock_attr);
    pthread_rwlock_destroy(&rwlock);
    return ret ;
}
```

创建100个线程（50个读线程和50个写线程），读线程只读取share_cnt的值，而写线程会将share_cnt的值自加。由于是while循环，所以属于读写竞争非常激烈的情况。创建线程之前，主线程会持有读写锁，直到所有线程创建完毕，然后主线程解锁，开闸放水，放任100个线程激烈地竞争读写锁。

如果采用读者优先的策略，则会看到由于读线程源源不断地申请读锁，写锁被活活饿死，写线程根本捞不到机会执行。运行N秒之后，share_cnt仍然是0。

如果我们采用写者优先的策略，情况就完全相反了，自从第一个写锁请求拿到锁之后，读锁请求就再也拿不到锁了，原因是总是有写锁请求，而写锁释放的时候，总是先唤醒写锁，表现出来很强大的惯性。

那么能否实现一款公平的读写锁呢？答案是肯定的。Locklessinc.com中有一篇题为《Sleeping Read-Write Locks》^[1]，在分析glibc实现的基础上，给出了一种公平的实现读写锁的方法，测试下来效率很不错。对锁的实现感兴趣的话，可以阅读该文章。

^[1] 《Sleeping Reader-Writer Lock》，请参见http://locklessinc.com/articles/sleeping_rwlocklocks/。

7.8.3 读写锁总结

从宏观意义上看，读写锁要比互斥量并发性好，因为读写锁在更多的时间区域内允许并发。

如果认为读写锁是完美的，以至于认为互斥锁没有存在的必要，那就是too young, too simple, sometimes naive了。Bryan Cantrill和Jeff Bonwick在《Real-world Concurrency》中提出的并发编程的建议里提到了要警惕读写锁（Be wary of readers-writer locks）。读写锁存在如下的短处。

- 性能：如果临界区比较大，读写锁高并发的优势就会显现出来，但是如果临界区非常小，读写锁的性能短板就会暴露出来。由于读写锁无论是加锁还是解锁，首先都会执行互斥操作，加上读写锁还需要维护当前读者线程的个数、写锁等待线程的个数、读锁等待线程的个数，因此这就决定了读写锁的开销不会小于互斥量。

- 饿死：互斥量虽然不是绝对意义上的公正，但是线程不会饿死。但是如上一小节的讨论，读者优先的策略下，写线程可能会饿死。写者优先的情况下，读线程可能会饿死。

- 死锁：读锁是可重入的，这就可能会引发死锁。考虑如下场景，读写锁采用写者优先的策略，A线程已经持有读锁，B线程申请了写锁，正处于等待状态，而持有读锁的A线程再次申请读锁，就会发生死锁。

比较适合读写锁的场景是：临界区的大小比较可观，绝大多数情况下是读，只有非常少的写。

7.9 性能杀手：伪共享

通过对互斥量和读写锁的讨论，我们已经有了这种意识：对于共享数据的读写，要加锁保护。临界区的存在，导致多个线程不能并行，造成性能下降。临界区越大，多个线程出入临界区越频繁，对性能的伤害也就越大。

这种情况下对性能的伤害是比较明显的。多线程情况下，还有一种情况对性能的损害是比较大的，却不像临界区这么明显。这就是有名的伪共享问题。

根据局部性原理，存储器是分层的，如图7-21所示。从距离CPU最近的寄存器到主内存，依次为CPU寄存器、L1 Cache、L2 Cache、L3 Cache和主存。从高层往底层走，存储设备变得更慢，容量更大，单位字节也更便宜。最高层是很少量的寄存器，通常可以在1个时钟周期内访问它们，而接下来的L1 Cache通常可以在4个时钟周期内访问到，L2 Cache通常需要10个时钟周期才能访问到，而到了主存，通常需要几百个时钟周期才能访问得到，对这个延迟数据感兴趣的话，可以阅读一下相关文献 [1][2]。

在这种分层的存储结构中，对于每一个k，位于k层的更快更小的存储被作为位于k+1层的更大更慢的存储设备的缓存。换句话说更快更小的存储设备的数据来自更慢更大的低一级存储设备。访问的数据在高速缓存中，被称为缓存命中，这种情况下访问速度比较快。如果访问的数据d在k级缓存中不存在，就不得不从k+1级中取出包含d的那个块（block）。如果k级缓存已经满了的话，就可能会覆盖现存的一个块。

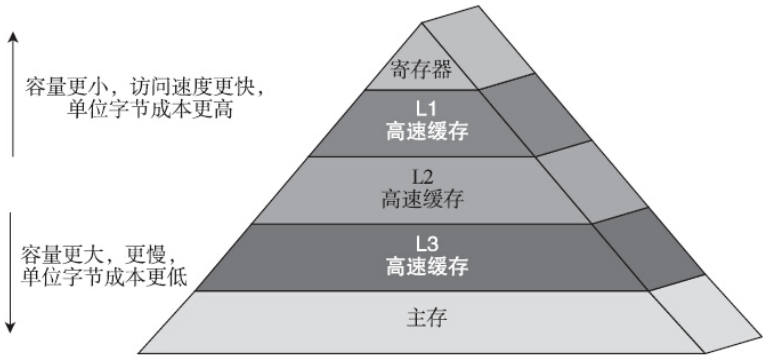


图7-21 存储器的层次结构

由于高一级缓存的性能远远超过低一级的缓存，所以一旦缓存不命中（Cache miss），对性能的损害就会是比较大的。

在典型的多核架构中，每个CPU都有自己的Cache。如果一个内存中的变量在多个CPU Cache中都有副本，则需要保证变量的Cache的一致性。现在大多数的架构实现Cache一致性都是采用MESI协议。对缓存一致性协议感兴趣的话，可以阅读《计算机体系结构：量化研究方法》这本经典之作。此外，Paul E.McKenney的《Is Parallel Programming Hard, And, If so, What Can You Do About It》一书中也有很详尽的介绍。

需要注意的是，CPU Cache是以缓存线（Cache line）为单位进行读写的。通常来说，一条缓存线的大小为64字节。换言之，就是访问1字节的数据，系统也会将该字节所在的整条缓存线的数据都搬到缓存中。

因为CPU Cache具有以Cache line为单位进行读写的性质，所以在多线程编程中，稍有不慎，就会掉入伪共享的陷阱，造成性能恶化。

可以考虑下如下代码：

```
int sum1;
int sum2;
void thread1(int v[], int v_count) {sum1 = 0;for (int i = 0; i < v_count; i++)      sum1 += v[i];
}
void thread2(int v[], int v_count) { sum2 = 0; for (int i = 0; i < v_count; i++)      sum2 += v[i];
}
```

这部分代码定义了两个全局变量sum1和sum2，两个线程分别将计算结果放入各自的全局变量中，看起来并行不悖。但是由于这两个全局变量紧挨着定义，编译器给这两个变量分配的内存几乎总是紧挨着的，因此这两个变量很可能在同一条Cache line中。

如图7-22所示，尽管线程1所在的CPU并不需要sum2的值，但是由于sum2和sum1在同一条Cache line中，因此sum2的值也随同sum1一并被加载到了thread1所在CPU的Cache中了。

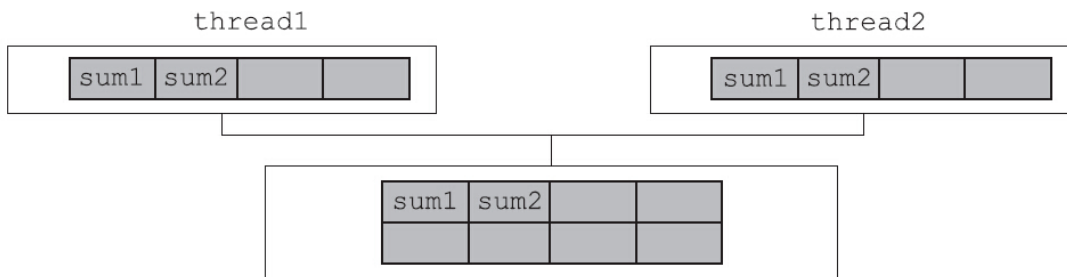


图7-22 伪共享

当thread1修改sum1的值时，尽管并未更新sum2的值，但影响的是整条Cache line，它会将thread2所在CPU对应的Cache line置为Invalidate。如果thread2尝试更新sum2，会触发缓存不命中。反过来，thread2修改sum2时，也会影响到sum1的缓存命中。

可以想见，就因为两个值彼此毗邻，落在同一条Cache line中，会导致大量的缓存不命中，从而影响性能。

下面通过一个例子，来看伪共享给性能带来的影响。

计算圆周率 π 有一种方法是数值积分法：

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

可以通过基于中点矩形的数值积分方法来求解上述积分，如下：

```
static long num_rect = 400000000;
double mid = 0.0;
double height = 0.0;
double width = 1.0/((double)num_rect);
int cur ;
for(cur = 0;cur < num_rect; cur += 1)
{
    mid = (cur+0.5)*width;
    height = 4.0/(1 + mid*mid);
    sum += height;
}
sum *= width;
```

这是典型的计算密集型程序，因此我们采用多线程来分工协作，代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#define NR_THREAD 1
static long num_rect = 400000000;
typedef struct sum_struct {
    double sum;
    //char padding[8];
} sum_struct;
struct sum_struct __sum[NR_THREAD];
void* calc_pi(void* ptr)
{
    int index = (int)ptr;
    double mid = 0.0;
    double height = 0.0;
    double width = 1.0/((double)num_rect);
    int cur = index;
    for(;cur < num_rect; cur += NR_THREAD)
    {
        mid = (cur+0.5)*width;
        height = 4.0/(1 + mid*mid);
        __sum[index].sum += height;
    }
    __sum[index].sum *= width;
}
int main()
{
    int i = 0;
    int ret ;
    double result = 0.0;
    pthread_t tid[NR_THREAD];
    fprintf(stdout,"The size of struct sum_struct = %ld\n",sizeof(struct sum_struct));
    for( i = 0 ; i < NR_THREAD; i++)
    {
        __sum[i].sum = 0.0;
        ret = pthread_create(&tid[i],NULL,calc_pi,(void*) i);
        if(ret != 0)
        {
            /*error handle here*/
            exit(1)
        }
    }
    for( i = 0; i < NR_THREAD ; i++)
    {
        pthread_join(tid[i],NULL);
        result += __sum[i].sum;
    }
    fprintf(stdout,"the PI = %.32f\n",result);
    return 0;
}
```

因为num_rect等于4亿，因此要计算4亿次，可以通过修改NR_THREAD的值，让8个线程协同计算，最后将结果累加到一起得到正确的值，希

望这样能将执行时间缩短为单线程的1/8，如图7-23所示。

线程 0	0	8	16
线程 1	1	9	17
⋮					
线程 7	7	15	23

图7-23 8个线程并发计算的值

因为每个线程都要负责往__sum对应的位置更新结果。因此这个数组很容易触发前面提到的伪共享陷阱。当sum_struct结构体没有填充字符时，该结构体占据8字节，当8个线程并发时，__sum数组很可能在同一个Cache line中，这时候性能必然会受到影响。为了避开false sharing这个陷阱，测试程序采用了加填充字节的方法。如果给sum_struct结构体加上56个填充字节，每个sum_struct占据1条Cache line的大小，则可以确保它们之间不会互相影响。

```
typedef struct sum_struct {
    double sum ;/*padding 56字节, 占满

1条

Cache line*/
    //char padding[56];
} sum_struct;
struct sum_struct __sum[NR_THREAD];
```

在24核的服务器上运行，结果如表7-14所示。

表7-14 伪共享测试代码的运行结果

测 试 场 景	运 行 时 间		
	real	user	sys
1 个线程	0m10.306s	0m10.308s	0m0.004s
8 个线程（没有 padding）	0m6.663s	0m49.976s	0m0.004s
8 个线程（padding 56 字节）	0m1.297s	0m10.324s	0m0.008s

可以看出，如果不加56字节的填充，由于伪共享引起的大量缓存不命中，8个线程并没有带来8倍的效率提升。通过填充字节解决了伪共享的问题之后，效率线性地提升了8倍。

[1] http://www.sisoftware.net/?d=qa&f=ben_mem_latency。
[2] Latency Number Every Programmer Should Know。

7.10 条件等待

条件等待是线程间同步的另一种方法。

线程经常遇到这种情况：要想继续执行，可能要依赖某种条件。如果条件不满足，它能做的事情就是等待，等到条件满足为止。通常条件的达成，很可能取决于另一个线程，比如生产者-消费者模型。当另外一个线程发现条件符合的时候，它会选择一个时机去通知等待在这个条件上的线程。有两种可能性，一种是唤醒一个线程，一种是广播，唤醒其他线程。

就像工厂里生产车间没有原料了，所有生产车间都停工了，工人们都在车间睡觉。突然进来一批原料，如果原料充足，你会发广播给所有车间，原料来了，快来开工吧。如果进来的原料很少，只够一个车间开工的，你可能只会通知一个车间开工。

为什么要有条件等待？考虑生产者-消费者模型，如果任务队列处于空的状态，那么消费者线程就应该停工等待，一直等到队列不空为止。如果没有条件等待，那么消费者线程的代码可能会写成这样：

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int WaitForTrue()
{
    pthread_mutex_lock(&m);
    while (condition is false) //条件不满足

{
    pthread_mutex_unlock(&m); //解锁等待其他线程改变共享数据

    sleep(n); //睡眠

n秒后再次加锁验证条件是否满足

    pthread_mutex_lock(&m);
}
}
```

如果条件不满足，就只能睡眠。上面的代码虽然也能满足这个要求，但存在严重的效率问题。考虑如下场景：解锁之后，sleep之前，等待的条件突然满足了，但很不幸，该线程仍然会睡眠n秒。

很自然需要这么一种机制：线程在条件不满足的情况下，主动让出互斥量，让其他线程去折腾，线程在此处等待，等待条件的满足；一旦条件满足，线程就可以立刻被唤醒。线程之所以可以安心等待，依赖的是其他线程的协作，它确信会有一个线程在发现条件满足以后，将向它发送信号，并且让

出互斥量。如果其他线程不配合（不发信号，不让出互斥量），这个主动让出互斥量并等待事件发生的线程就真的要等到花儿都谢了。

7.10.1 条件变量的创建和销毁

NPTL使用pthread_cond_t类型的变量来表示条件变量。条件变量不是一个值，我们无法给条件变量赋值。一个线程如果要等待某个事件的发生，或者某个条件的满足，那么这个线程需要的是条件变量：线程等待在条件变量上。

和互斥锁一样，条件变量在使用之前要先初始化。互斥锁有静态初始化，条件变量也一样。简单地把PTHREAD_COND_INITIALIZER赋值给pthread_cond_t类型的变量就可完成条件变量的初始化：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

动态分配条件变量，或者对条件变量的属性有所定制，都需要用pthread_cond_init进行初始化：

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

如果采用默认属性，可以将NULL作为第二个参数。

对于pthread_cond_init初始化的条件变量，不要忘记调用pthread_cond_destroy来销毁。其接口定义如下：

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

对于条件变量的初始化和销毁，需要注意以下几点：

- 永远不要用一个条件变量对另一个条件变量赋值，即pthread_cond_t cond_b=cond_a不合法，这种行为是未定义的。

- 使用PTHREAD_COND_INITIALIZER静态初始化的条件变量，不需要被销毁。

- 要调用pthread_cond_destroy销毁的条件变量可以调用pthread_cond_init重新进行初始化。

- 不要引用已经销毁的条件变量，这种行为是未定义的。

有了条件变量的初始化和销毁，就可以进入正题了。接下来看看如何使用条件变量。

7.10.2 条件变量的使用

条件变量，天生就是与条件的满足与否相伴而生的。通常，线程会对一个条件进行测试，如果条件不满足，就等待（`pthread_cond_wait`），或者等待一段有限的时间（`pthread_cond_timedwait`）。相关函数的定义如下：

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
```

从接口上可以看出，条件等待总是和互斥量绑定在一起的。为什么要这样设计？

条件等待是线程间同步的一种手段，如果只有一个线程，条件不满足，那么等待千年也是枉然，所以必须要有一个线程通过某些操作，改变共享数据，使原先不满足的条件变得满足了，并且友好地通知等待在条件变量上的线程。

条件不会无缘无故地突然变得满足了，必然会牵扯到共享数据的变化。所以一定要有互斥锁来保护。没有互斥锁，就无法安全地获取和修改共享数据。

好吧，就算如此，先调用`pthread_mutex_lock`，发现条件不满足，解锁，然后等待在条件上就行了，为什么还要把互斥锁作为参数传给`pthread_cond_wait`呢？像下面所示代码这样使用不可以吗？

// 错误的设计

```
pthread_mutex_lock(&m)
while(condition_is_false)
{
    pthread_mutex_unlock(&m);
    // 解锁之后，等待之前，可能条件已经满足，信号已经发出，但是该信号可能会被错过

    cond_wait(&cv);
    pthread_mutex_lock(&m);
}
```

原因在于，上面的解锁和等待不是原子操作。解锁以后，调用`cond_wait`之前，如果已经有其他线程获取到了互斥量，并且满足了条件，同时发出了通知信号，那么`cond_wait`将错过这个信号，可能会导致线程永远处于阻塞状态。所以解锁加等待必须是一个原子性的操作，以确保已经注册到事件的等待队列之前，不会有其他线程可以获得互斥量。

那先注册等待事件，后释放锁不行吗？注意，条件等待是个阻塞型的接口，不单单是注册在事件的等待队列上，线程也会因此阻塞于此，从而导致互斥量无法释放，其他线程获取不到互斥量，也就

无法通过改变共享数据使等待的条件得到满足，因此这就造成了死锁。

下面的伪代码显示了POSIX如何使用条件变量v和互斥量m来等待条件的发生：

```
pthread_mutex_lock(&m);
while(condition_is_false)
    pthread_cond_wait(&v, &m); // 此处会阻塞
```

```
/* 如果代码运行到此处，则表示我们等待的条件已经满足了，
```

```
    * 并且在此持有了互斥量
```

```
*/
/* 在满足条件的情况下，做你想做的事情。
```

```
*/
pthread_mutex_unlock(&m);
```

pthread_cond_wait函数只能由拥有互斥量的线程来调用，当该函数返回的时候，系统会确保该线程再次持有互斥量，所以这个接口容易给人一种误解，就是该线程一直在持有互斥量。事实上并不是这样的。这个接口向系统声明了我的心在等待，永远在等待之后，就把互斥量给释放了。这样其他线程就有机会持有互斥量，操作共享数据，触发变化，使线程等待的条件得到满足。

既然互斥量和条件变量关系如此紧密，为什么不干脆将互斥量变成条件变量的一部分呢？原因是，同一个互斥量上可能有不同的条件变量，比如说，有的线程希望队列不空的时候发送信号，有的线程希望队列满的时候发送通知给它（为了创建更多的线程做消费者或其他目的）。

pthread_cond_timedwait函数与pthread_cond_wait的工作方式几乎是一样的，只是调用时需要指定一个超时的时间。注意这个时间是绝对时间，而不是相对时间。如果最多等待2分钟，那么这个值应该是当前时间加上2分钟。

上面将互斥量和条件变量配合使用的示范代码中有个很有意思的地方，就是用了while语句，醒来之后要再次判断条件是否满足。

```
while(condition_is_false)
    pthread_cond_wait(&v, &m); // 此处会阻塞
```

为什么不写成：

```
if(condition_is_false)
    pthread_cond_wait(&v, &m); //此处会阻塞
```

唤醒以后，再次检查条件是否满足，是不是多此一举？

答案是不得不如此。因为唤醒中存在虚假唤醒（spurious wakeup），换言之，条件尚未满足，`pthread_cond_wait`就返回了。在一些实现中，即使没有其他线程向条件变量发送信号，等待此条件变量的线程也有可能会醒来。

看起来这像是个bug，但它是实实在在存在的。为什么会存在虚假唤醒？一个原因是`pthread_cond_wait`是futex系统调用，属于阻塞型的系统调用，当系统调用被信号中断的时候，会返回-1，并且把`errno`置为EINTR。很多这种系统调用为了防止被信号中断都会重启系统调用，代码如下：

```
pid_t r_wait(int *stat_loc)
{
    int retval;
    while(((retval = wait(stat_loc)) == -1) && (errno == EINTR));
    return retval;
}
```

但是futex不一样，在futex返回之后，到重启系统调用之前，可能已经调用过`pthread_cond_signal`或`pthread_cond_broadcast`。一旦错失，再次调用`pthread_cond_wait`可能会导致无限制地等待下去。为了防止这种情况，宁可虚假唤醒，也不能再次调用`pthread_cond_wait`，以免陷入无穷的等待中。

除了上面的信号因素外，还存在以下情况：条件满足了发送信号，但等到调用`pthread_cond_wait`的线程得到CPU资源时，条件又再次不满足了。好在无论是哪种情况，醒来之后再次测试条件是否满足就可以解决虚假等待的问题。

条件等待，等于把控制权交给了别的线程，它信任别的线程会在合适的时机通知它，这是多大的信任啊。如果其他线程忘记发送信号了，那么条件等待的线程就彻底“悲剧”了。

如何发送信号来通知等待的线程呢？POSIX提供了如下两个接口：

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

`pthread_cond_signal`负责唤醒等待在条件变量上的一个线程，`pthread_cond_broadcast`，顾名思义，就是广播唤醒等待在条件变量上的所有线程。

等一下，刚才讲解`pthread_cond_wait`的时候曾提到过，线程醒来时会确保持有互斥量，为何广播还

能唤醒等待在条件变量上的所有线程呢，不是前后矛盾吗？

答案是不矛盾，所有的线程被广播唤醒了之后，集体争夺互斥锁，没抢到的继续睡。从内核中醒来，然后继续睡去，是一种性能的浪费。

使用通知机制来完成线程同步，代码范例如下：

// 为了让流程更加清晰，此处忽略了

```
error handle
pthread_mutex_lock(&m);
/*一些对共享数据的操作，会导致另一个线程等待的条件满足
```

```
*/
//此处也可以是
```

```
pthread_cond_broadcast函数
```

```
pthread_cond_signal(&cond);
pthread_mutex_unlock(&m);
```

发送信号，通知等待在条件上的线程，然后解锁互斥量。

注意范例代码中先发送信号，然后解锁互斥量，这个顺序不是必须的，也可以颠倒。标准允许任意顺序执行这两个调用。

有什么区别吗？

先通知条件变量、后解锁互斥量，效率会比先解锁、后通知条件变量低。因为先通知后解锁，执行pthread_cond_wait的线程可能在互斥量已然处于加锁状态的时候醒来，发现互斥量仍然没有解锁，就会再次休眠，从而导致了多余的上下文切换。某些实现使用等待变形（wait morphing）来优化这个问题：并不真正地唤醒执行pthread_cond_wait的线程，而是将线程从条件变量的等待队列转移到互斥量的等待队列上，从而消除无谓的上下文切换。

glibc对pthread_cond_broadcast做了类似的优化，即只唤醒一个线程，将其他线程从条件变量的等待队列搬移到了互斥量的等待队列中。对实现细节感兴趣的可以参阅Ulrich Drepper的《Futexes Are Tricky》。

先解锁、后通知条件变量虽然可能会有性能上的优势，但是也会带来其他的问题。如果存在一个高优先级的线程，既等待在互斥量上，也等待在条件变量上；同时还存在一个低优先级的线程，只等

待在互斥量上。一旦先解锁互斥量，低优先级的进程就可能会抢先获得互斥量，待调用 `pthread_cond_signal` 之后，高优先级的进程会发现互斥量已经被低优先级的进程抢走了。

第8章 理解Linux线程（2）

第7章介绍了线程的基本接口，这些基本接口非常重要，掌握了这些基本接口，就能应对绝大多数的应用场景。本章将介绍一些线程相关的其他内容。

8.1 线程取消

线程可以通过调用`pthread_cancel`函数来请求取消同一进程中的其他线程。

从编程的角度来讲，不建议使用这个接口。笔者对该接口的评价不高，该接口实现了一个似是而非的功能，却引入了一堆问题。陈硕在《Linux多线程服务器编程》一书中也提到过，不建议使用取消接口来使线程退出，个人表示十分赞同。

8.1.1 函数取消接口

Linux提供了如下函数来控制线程的取消：

```
int pthread_cancel(pthread_t thread);
```

一个线程可以通过调用该函数向另一个线程发送取消请求。这不是个阻塞型接口，发出请求后，函数就立刻返回了，而不会等待目标线程退出之后才返回。

如果成功，该函数返回0，否则将错误码返回。

对于glibc实现而言，调用pthread_cancel时，会向目标线程发送一个SIGCANCEL的信号，该信号就是6.4节“信号的分类”中提到的被NPTL征用的32号信号。

线程收到取消请求后，会采取什么行动呢？这取决于该线程的设定。NPTL提供了函数来设置线程是否允许取消，以及在允许取消的情况下，如何取消。

pthread_setcancelstate函数用来设置线程是否允许取消，函数定义如下：

```
int pthread_setcancelstate(int state, int *oldstate);
```

state参数有两种可能的值：

·PTHREAD_CANCEL_ENABLE？

·PTHREAD_CANCEL_DISABLE

如果取消状态是PTHREAD_CANCEL_DISABLE，则表示线程不理睬取消请求，取消请求会被暂时挂起，不予处理。

线程的默认取消状态是PTHREAD_CANCEL_ENABLE。如果state是PTHREAD_CANCEL_ENABLE，那么收到取消请求后，会发生什么？这取决于线程的取消类型。

pthread_setcanceltype函数用来设置线程的取消类型，其定义如下：

```
int pthread_setcanceltype(int type, int *oldtype);
```

取消类型有两种值：

·PTHREAD_CANCEL_DEFERRED

·PTHREAD_CANCEL_ASYNCHRONOUS

PTHREAD_CANCEL_ASYNCHRONOUS为异步取消，即线程可能在任何时间点（可能是立即取消，但也不一定）取消线程。这种取消方式的最大问题在于，你不知道取消时线程执行到了哪一步。所以，这种取消方式太粗暴，很容易造成后续的混乱。因此不建议使用该取消方式。

PTHREAD_CANCEL_DEFERRED是延迟取消，线程会一直执行，直到遇到一个取消点，这种方式也是新建线程的默认取消类型。

什么是取消点？就是对于某些函数，如果线程允许取消且取消类型是延迟取消，并且线程也收到了取消请求，那么当执行到这些函数的时候，线程就可以退出了。

标准规定了很多函数必须是取消点，由于太多（有好几十个之多），就不一一罗列了，通过man pthreads可以查询到这些取消点函数。

线程执行到取消点，会自动处理取消请求，但是如果线程没有用到任何取消点函数，那该怎么办，如何响应取消请求？

为了应对这种场景，系统引入了pthread_testcancel函数，该函数一定是取消点。所以编程者可以周期性地调用该函数，只要有取消请求，线程就能响应。该函数定义如下：

```
void pthread_testcancel(void);
```

如果线程被取消，并且其分离状态是可连接的，那么需要由其他线程对其进行连接。连接之后，pthread_join函数的第二个参数会被置成PTHREAD_CANCELED，通过该值可以知道线程并不是“寿终正寝”，而是被其他线程取消而导致的退出。

接口都介绍完了，是时候讨论下线程取消的弊端了。线程取消是一种在线程的外部强行终止线程的执行做法，由于无法预知目标线程内部的情况，尤其是第一种异步取消类型，因此可能会带来毁灭性的结果。

目标线程可能会持有互斥量、信号量或其他类型的锁，这时候如果收到取消请求，并且取消类型是异步取消，那么可能目标线程掌握的资源还没有来得及释放就被迫退出了，这可能会给其他线程带来不可恢复的后果，比如死锁（其他线程再也无法获得资源）。

即使执行异步取消也安然无恙的函数称为异步取消安全函数（async-cancel-safe function），手册里说只有下述三个函数是异步取消安全函数，所以对于其他函数，一律都不是异步取消安全函数。

```
pthread_cancel()  
pthread_setcancelstate()  
pthread_setcanceltype()
```

所以对编程人员而言，应该遵循以下原则：

·第一，轻易不要调用`pthread_cancel`函数，在外部杀死线程是很糟糕的做法，毕竟如果想通知目标线程退出，还可以采取其他方法。

·第二，如果不得不允许线程取消，那么在某些非常关键不容有失的代码区域，暂时将线程设置成不可取消状态，退出关键区域之后，再恢复成可以取消的状态。

·第三，在非关键的区域，也要将线程设置成延迟取消，永远不要设置成异步取消。

8.1.2 线程清理函数

假设遇到取消请求，线程执行到了取消点，却没有来得及做清理动作（如动态申请的内存没有释放，申请的互斥量没有解锁等），可能会导致错误的产生，比如死锁，甚至是进程崩溃。

下面来看一个简单的例子：

```
void* cancel_unsafe(void*) {
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&mutex);          // 此处不是撤消点

    struct timespec ts = {3, 0};
    nanosleep(&ts, 0);                  // 是撤消点

    pthread_mutex_unlock(&mutex);        // 此处不是撤消点

    return 0;
}

int main(void) {
    pthread_t t;
    pthread_create(&t, 0, cancel_unsafe, 0);
    pthread_cancel(t);
    pthread_join(t, 0);
    cancel_unsafe(0); // 发生死锁!

    return 0;
}
```

在上面的例子中，`nanosleep`是取消点，如果线程执行到此处时被其他线程取消，就会出现以下情况：互斥量还没有解锁，但持有锁的线程已不复存在。这种情况下其他线程再也无法申请到互斥量，很有可能在某处就会陷入死锁的境地。

为了避免这种情况，线程可以设置一个或多个清理函数，线程取消或退出时，会自动执行这些清理函数，以确保资源处于一致的状态。其相关接口定义如下：

```
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

标准允许用宏（`macro`）来实现这两个接口，Linux就是用宏来实现的。这意味着这两个函数必须同时出现，并且属于同一个语法块。

何为同一个语法块？比较难解释，我尝试来解释一下它的反面。如果两个函数在不同的函数中出现，它们就不是处于同一个语法块。示例代码如下：

```
void foo()
{
    .....
    pthread_cleanup_pop(0)
    .....
}
void *thread_work(void *arg)
{
    .....
    pthread_cleanup_push(clean, clean_arg);
    .....
    foo();
    .....
}
```

这个例子比较简单，因为`pthread_cleanup_push`在线程的主函数里面，而`pthread_cleanup_pop`在另外一个函数里面，这一对函数明显不在一个语法块里面。

上面这种错误是很好防范的，比较难防范的是下面这种：

```
pthread_cleanup_push(clean_func, clean_arg);
.....
if (cond)
{
    pthread_cleanup_pop(0);
}
```

在日常编码中很容易犯上面这种错误。因为`pthread_cleanup_push`和`pthread_cleanup_pop`的实现中包含了`{`和`}`，所以将`pop`放入`if{}`的代码块中，会导致括号匹配错乱，最终会引发编译错误。

第二个需要注意的是，可以注册多个清理函数，如下所示：

```
pthread_cleanup_push(clean_func_1, clean_arg_1)
pthread_cleanup_push(clean_func_2, clean_arg_2)
...
pthread_cleanup_pop(execute_2);
pthread_cleanup_pop(execute_1);
```

从`push`和`pop`的名字可以看出，这是栈的风格，后入先出，就是后注册的清理函数会先执行。

其中`pthread_cleanup_pop`的用处是，删除注册的清理函数。如果参数是非0值，那么执行一次，再删除清理函数。否则的话，就直接删除清理函数。

第三个问题最关键，何时会触发注册的清理函数：

- 当线程的主函数是调用`pthread_exit`返回的，清理函数总是会被执行。
- 当线程是被其他线程调用`pthread_cancel`取消的，清理函数总是会被执行。
- 当线程的主函数是通过`return`返回的，并且`pthread_cleanup_pop`的唯一参数`execute`是0时，清理函数不会被执行。
- 当线程的主函数是通过`return`返回的，并且`pthread_cleanup_pop`的唯一参数`execute`是非零值时，清

理函数会执行一次。

下面看下示例代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<string.h>
void clean(void* arg)
{
    printf("CLEAN_UP:%s\n", (char*)arg);
}
void *thread(void *param)
{
    int input = (int)param;
    printf("thread start\n");
    pthread_cleanup_push(clean, "first cleanup handler");
    pthread_cleanup_push(clean, "second cleanup handler");
    /*work logic here*/
    if(input != 0){
        /*pthread_exit退出, 清理函数总会被执行

*/
        pthread_exit((void*)1);
    }
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    /*return 返回, 如果上面

pop函数的参数是

0, 则不会执行清理函数

*/
    return ((void *)0);
}
int main()
{
    pthread_t tid ;
    void *res ;
    int ret ;
    ret = pthread_create(&tid, NULL, thread, (void*)0);
    if(ret != 0)
    {
        /*error handle here*/
        return -1;
    }
    pthread_join(tid, &res);
    printf("first thread exit, return code is %d\n", (int)res);
    ret = pthread_create(&tid, NULL, thread, (void*)1);
    if(ret != 0)
    {
        /*error handle here*/
        return -1;
    }
    pthread_join(tid, &res);
    printf("second thread exit, return code is %d\n", (int)res);
    return 0;
}
```

当线程用return退出，并且pthread_cleanup_pop的参数是0时，那么注册的清理函数不被执行：

```
thread start
first thread exit, return code is 0
thread start
CLEAN_UP:second cleanup handler
CLEAN_UP:first cleanup handler
second thread exit, return code is 1
```

如果将上面示例代码中的`pthread_cleanup_pop`的参数改成1，就会发现，无论是调用`pthread_exit`函数返回，还是在线程的主函数中调用`return`返回，都会调用清理函数：

```
thread start
CLEAN UP:second cleanup handler
CLEAN UP:first cleanup handler
first thread exit,return code is 0
thread start
CLEAN UP:second cleanup handler
CLEAN UP:first cleanup handler
second thread exit,return code is 1
```

有了清理函数，本节开头处提到的例子就可以改进为如下形式了：

```
void cleanup(void* mutex) {
    pthread_mutex_unlock((pthread_mutex_t*)mutex);
}
void* cancel_unsafe(void*) {
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_cleanup_push(cleanup, &mutex);

    pthread_mutex_lock(&mutex);
    struct timespec ts = {3, 0};
    nanosleep(&ts, 0);
    pthread_mutex_unlock(&mutex);
    pthread_cleanup_pop(0);

    return 0;
}
```

在这种情况下，如果线程被取消，清理函数则会负责解锁操作。

8.2 线程局部存储

`errno`变量是线程局部存储的典型示例。我们可以通过该案例来理解引入线程局部存储的意义。

在多线程引入之前，由于进程只有一条控制流（暂不考虑信号处理函数），因此当函数调用出错时，可以通过设置全局的`errno`来提示遇到的错误类型。代码如下所示：

```
int f = open (...);
if (f < 0)
    printf ("error %d encountered\n", errno);
```

但是自从引入多线程之后，情况就发生了变化。如果`errno`仍然是进程内的全局变量，就会引起混乱。考虑如下两个线程分别执行如下代码：

```
线程

1
int f = open (...);
if (f < 0)
    printf ("error %d encountered\n", errno);线程

2
int s = socket (...);
if (s < 0)
    printf ("error %d encountered\n", errno);
```

当两个线程同时执行这两部分代码并且几乎同时出错的话，后一个出错时设置的`errno`的值会覆盖前一个出错时设置的`errno`。因此至少有一个输出的`errno`是不对的。

对于这个问题，一种解决的方法是这样的：

```
int local_errno
int f = open(...,&local_errno)
if (f < 0)
    printf ("error %d encountered\n", local_errno);
```

这种方法固然可以做到对多线程的支持，但是在现实中不具备可操作性。大量的函数接口已经存在很久，改变接口意味着不兼容历史代码。对`errno`而言，比较好的方案是既要能应对多线程，又不需要改变既有的接口。

这时候，线程局部存储就横空出世了。使用线程局部存储（Thread Local Storage）技术就能满足上述的需求。该技术为每一个线程都分别维护一个变量的副本，尽管名字相同却分别存储，并行不悖。

在Linux下有两种方法可以实现线程局部存储：

- 使用NPTL提供的函数。

- 使用编译器扩展的__thread关键字。

8.2.1 使用NPTL库函数实现线程局部存储

NTPL提供了一个函数接口来实现线程局部存储的功能：

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
int pthread_key_delete(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
void *pthread_getspecific(pthread_key_t key);
```

其中，`pthread_key_create`函数会为线程局部存储创建一个新键，并通过给`key`赋值，返回给用户使用。

因为进程中的所有线程都可以使用返回的键，所以参数`key`应该指向一个全局变量。

参数`destructor`指向一个自定义的函数：

```
void * destructor(void *value)
{
    /*多是为了释放

value指针指向的资源

*/
}
```

线程终止时，如果`key`关联的值不是NULL，那么NTPL会自动执行定义的`destructor`函数。如果无须解构，可以将`destructor`设置为NULL。

这几个接口比较晦涩，很难从接口上想到如何使用线程局部存储。下面通过一个例子来说明如何使用这些接口。在下面的例子里，程序希望每一个线程将自己的log输出到各自独立的文件。

```
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/* 用于为每个线程保存文件指针的
```

```
TSD 键值
```

```
, 根据键值可以找到线程各自的
```

```
Data.
```

```
*/
static pthread_key_t  thread_log_key;
void write_to_thread_log(const_char *message)
```

```

{
    FILE* thread_log=(FILE*)pthread_getspecific(thread_log_key);
    fprintf(thread_log, "%s\n", message);
}
void fn_close_thread_log(void *thread_log)
{
    fclose((FILE *)thread_log);
}
void *thread_function(void *args)
{
    char thread_log_filename[128];
    FILE *thread_log;sprintf(thread_log_filename, "thread%d.log",
        unsigned long)pthread_self());
    thread_log = fopen(thread_log_filename, "w");
    /* 将文件指针保存在

```

thread_log_key标识的

TSD 中。

```

*/
pthread_setspecific(thread_log_key,thread_log);
write_to_thread_log("Thread starting.");
return NULL;
}
int main()
{
    int i;
    pthread_t threads[5];
    /* 创建一个键值，用于将线程日志文件指针保存在

```

TSD中。

* 调用

close_thread_log以关闭这些文件指针。

```

*/
pthread_key_create(&thread_log_key, fn_close_thread_log);
for(i = 0; i < 5; ++i)
    pthread_create(&(threads[i]), NULL, thread_function, NULL);
for(i = 0; i < 5; ++i)
    pthread_join(threads[i], NULL);
return 0;
}

```

上面的程序首先调用pthread_key_create函数来申请一个槽位。在NPTL实现下，pthread_key_t是无符号整型，pthread_key_create调用成功时会将一个小于1024的值填入第一个入参指向的pthread_key_t类型的变量中。

为什么键值总是要小于1024？那是因为NPTL实现一共提供了1024个槽位。

如图8-1所示，记录槽位分配情况的数据结构pthread_keys是进程唯一的。对于上面的示例代码而言，第一次调用pthread_key_create毫无疑问会领到slot 0。即thread_log_key的值为0，表示占用了0号槽位，如图8-1所示。

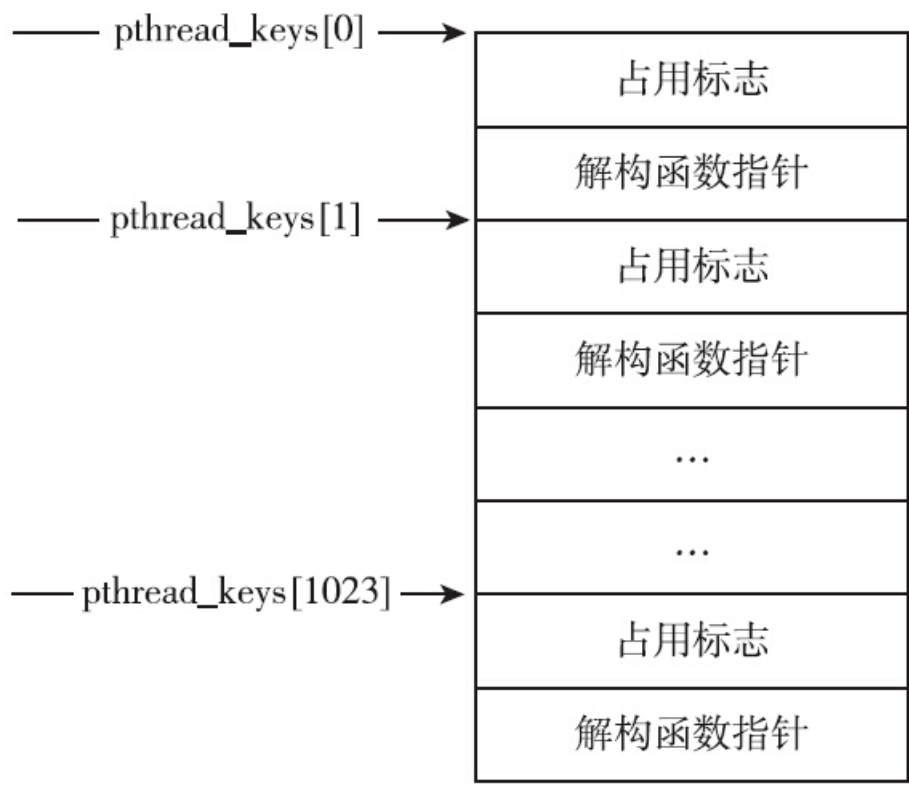


图8-1 pthread_keys与槽位分配

目前，各个线程还没有数据和该key相关联。接下来线程函数通过调用pthread_setspecific函数，将key分别与各自的线程数据关联起来。

```
pthread_setspecific(thread_log_key,thread_log);
```

每个线程槽位号0各自指向了线程自己的数据。从此处开始分家，key是同一个key，但每个线程指向的数据各不相同（如图8-2所示）。

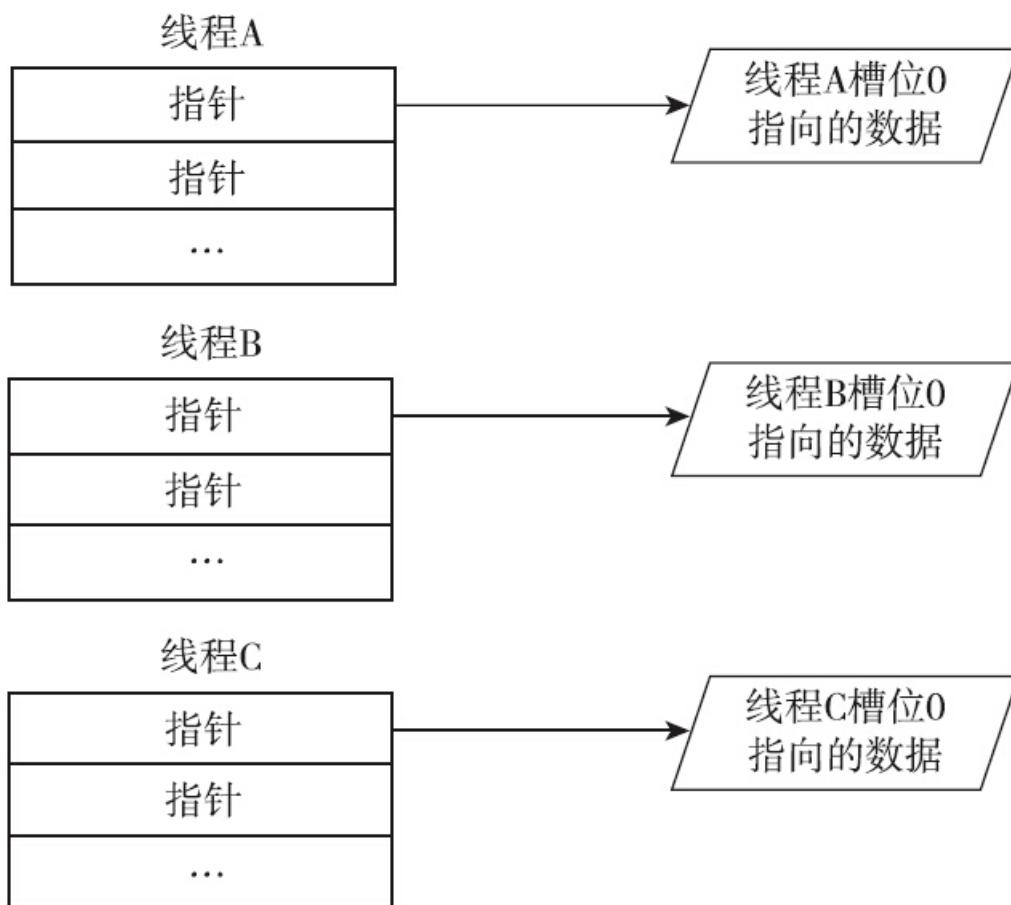


图8-2 同一个key每个线程指向各自的数据

线程如果想要使用各自的值怎么办？拿这个key，去找到与key关联的数据结构，这是pthread_getspecific函数的职责所在。

```
FILE* thread_log =(FILE *)pthread_getspecific(thread_log_key);
```

因为线程知道key关联的数据结构是什么类型，所以可以从key直接获取到key指向的value。取到线程的特有数据之后，就可以操作了。

由于key属于全局变量，因此取到的线程特有数据value就变成了线程内部的“全局变量”。

1024个key，对于普通的应用来说足够了。如果一个多线程应用确实需要很多的线程特有数据，那么可以将其封装在一个数据结构之内。

这种方法，允许的键值个数有限并不是问题的关键，问题的关键是它的接口太难用了，接口设计得有点反人类。

8.2.2 使用__thread关键字实现线程局部存储

由于8.2.1节提供的接口太难用，有人想到了在编译器中增加新功能，支持特定的关键字__thread，隐式地构造线程局部变量。

它的使用方法非常简单：

```
__thread int val = 0;
```

凡是带有__thread关键字的变量，每个线程都会有该变量的一个拷贝，并行不悖，互不干扰。该局部变量一直都在，直到线程退出为止。

使用线程局部变量需要注意以下几点：

- 如果变量生命中使用了关键字static或extern，那么关键字__thread应该紧随其后。
- 声明时，可以正常初始化。
- 可以通过取地址操作符（&）获取到线程局部变量的地址。

同样的例子，用__thread关键字来实现就自然多了，代码如下：

```
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
__thread FILE* thread_log = NULL ;
void write_to_thread_log(const char *message)
{
    fprintf(thread_log, "%s\n", message);
}
void *thread_function(void *args)
{
    char thread_log_filename[128];sprintf(thread_log_filename, "thread%d.log",
        (unsigned long)pthread_self());
    thread_log = fopen(thread_log_filename, "w");
    write_to_thread_log("Thread starting.");
    fclose(thread_log);
    return NULL;
}
int main()
{
    int i;
    pthread_t threads[5];
    for(i = 0; i < 5; ++i)
        pthread_create(&(threads[i]), NULL, thread_function, NULL);
    for(i = 0; i < 5; ++i)
        pthread_join(threads[i], NULL);
    return 0;
}
```

线程局部存储需要内核，Pthreads实现和C编译器提供了支持。对线程局部存储（Thread Local Storage）的实现感兴趣的话，Ulrich Drepper著有“ELF Handling For Thread-Local Storage”一文，是非常好的参考资料。

8.3 线程与信号

信号出现地要比线程早，所以设计信号时，尚没有线程。在引入线程之后，如何设计信号成了一个难点。既要保证传统的语义不变，又要设计出适用于多线程环境的信号模型，确实难度不小。

在第6章“信号”一章中，已基本讲清楚了多线程和信号的关系，以及内核如何实现。在此，仅仅总结一下：

- 信号处理函数是进程层面的概念，或者说是线程组层面的概念，线程组内所有线程共享对信号的处理函数。

- 对于发送给进程的信号，内核会任选一个线程来执行信号处理函数，执行完后，会将其从挂起信号队列中去除，其他进程不会对一个信号重复响应。

- 可以针对进程中的某个线程发送信号，那么只有该线程能响应，执行相应的信号处理函数。

- 信号掩码是线程层面的概念，信号处理函数在线程组内是统一的，但是信号掩码是各自独立可配置的，各个线程独立配置自己要阻止或放行的信号集合。

- 挂起信号（内核已经收到，但尚未递送给线程处理的信号）既是针对进程的，又是针对线程的。内核维护两个挂起信号队列，一个是进程共享的挂起信号队列，一个是线程特有的挂起信号队列。调用函数`sigpending`返回的是两者的并集。对于线程而言，优先递送发给线程自身的信号。

上面这些内容，基本概括了多线程条件下信号的模型。内核如何做到这些模型，在第6章中基本都有介绍，在此处就不再赘述了。

8.3.1 设置线程的信号掩码

前面已提到过，信号掩码是针对线程的，每个线程都可以自行设置自己的信号掩码。如果自己不上设置，就会继承创建者的信号掩码。

NPTL实现了如下接口来设置线程的信号掩码：

```
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
```

how的值用来指定如何更改信号组：

- SIG_BLOCK向当前信号掩码中添加new，其中new表示要阻塞的信号组。
- SIG_UNBLOCK从当前信号掩码中删除new，其中new表示要取消阻塞的信号组。
- SIG_SETMASK将当前信号掩码替换为new，其中new表示新的信号掩码。

该接口的使用方式和sigprocmask一模一样，在Linux上，两个函数的实现是相同的。



说明 SIGCANCEL和SIGSETXID信号被用于NPTL实现，因此用户不能也不应该改变这两个信号的行为方式。好在用户不用操心这两个信号，sigprocmask函数和pthread_sigmask函数对这两者都做了特殊处理。

8.3.2 向线程发送信号

第6章提到过向线程发送信号的系统调用`tkill/tgkill`，无奈`glibc`并未将它们封装成可以直接调用的函数。不过，幸好提供了另外一个函数：

```
int pthread_kill(pthread_t thread, int sig);
```

由于`pthread_t`类型的线程ID只在线程组内是唯一的，其他进程完全可能存在线程ID相同的线程，所以`pthread_kill`只能向同一个进程的线程发送信号。

除了这个接口外，Linux还提供了特有的函数将`pthread_kill`和`sigqueue`功能累加在一起：

```
#define _GNU_SOURCE
#include <pthread.h>
int pthread_sigqueue(pthread_t thread, int sig,
                    const union sigval value);
```

这个接口和`sigqueue`一样，可以发送携带数据的信号。当然，只能发给同一个进程内的线程。

8.3.3 多线程程序对信号的处理

单线程的程序，对信号的处理已经比较复杂了。因为信号打断了进程的控制流，所以信号处理函数只能调用异步信号安全的函数。而异步信号安全是个很苛刻的条件。

多线程的引入，加剧了这种复杂度。因为信号可以发送给进程，也可以发送给进程内的某一线程。不同线程还可以设置自己的掩码来实现对信号的屏蔽。而且，没有一个线程相关的函数是异步信号安全的，信号处理函数不能调用任何pthread函数，也不能通过条件变量来通知其他线程。

正如陈硕在《Linux多线程服务器编程》中提到的，在多线程程序中，使用信号的第一原则就是不要使用信号。

- 不要主动使用信号作为进程间通信的手段，收益和引入的风险完全不成比例。

- 不主动改变异常处理信号的信号处理函数。用于管道和socket的SIGPIPE可能是例外，默认语义是终止进程，很多情况下，需要忽略该信号。

- 如果无法避免，必须要处理信号，那么就采用sigwaitinfo或signalfd的方式同步处理信号，减少异步处理带来的风险和引入bug的可能。

在第6章中，曾经分析了如何使用sigwaitinfo函数和signalfd同步地处理信号，此处就不再赘述了。

8.4 多线程与fork（）

多线程和fork函数的协作性非常差。对于多线程和fork，最重要的建议就是永远不要在线程程序里面调用fork。

请跟我再念一遍：永远不要在线程程序里面调用fork。

Linux的fork函数，会复制一个进程，对于多线程程序而言，fork函数复制的是调用fork的那个线程，而并不复制其他的线程。fork之后其他线程都不见了。Linux不存在forkall语义的系统调用，无法做到将多线程全部复制。

多线程程序在fork之前，其他线程可能正持有互斥量处理临界区的代码。fork之后，其他线程都不见了，那么互斥量的值可能处于不可用的状态，也不会有其他线程来将互斥量解锁。

下面用一个例子来描述这种场景：

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/wait.h>
static void* worker(void* arg)
{
    pthread_detach(pthread_self());
    for (;;)
    {
        setenv("foo", "bar", 1);
        usleep(100);
    }
    return NULL;
}
static void sigalrm(int sig)
{
    char a = 'a';
    write(fileno(stderr), &a, 1);
}
int main()
{
    pthread_t setenv_thread;
    pthread_create(&setenv_thread, NULL, worker, 0);
    for (;;)
    {
        pid_t pid = fork();
        if (pid == 0)
        {
            signal(SIGALRM, sigalrm);
            alarm(1);
            unsetenv("bar");
            exit(0);
        }
        wait3(NULL, WNOHANG, NULL);
        usleep(2500);
    }
    return 0;
}
```

上面的代码比较简单，创建了一个线程周期性地执行setenv函数，修改环境变量。主线程会fork子进程，子进程负责执行unsetenv函数，同时调用了alarm，一秒钟后会收到SIGALRM信号。子进程通过执行signal函数，注册了SIGALRM信号的处理函数，即向标准错误打印字母‘a’。

fork创建的子进程在调用alarm注册的闹钟之后，只执行unsetenv函数，然后就会调用exit退出。因此，在正常情况子进程很快就会退出，alarm约定的1秒钟时间还未到就退出了。也就是说，信号处理函数不应该被执行，自然也就不应该打印出字母‘a’。

可是实际情况是：

```
./thread_fork
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa^C
```

原因何在？在某些情况下，子进程为什么不能及时退出，以至于过了1秒之后，子进程还没有退出？

选择一个阻塞的线程，用gdb调试下，看看到底阻塞在何处。

```
(gdb) bt
#0  _l1l_lock_wait_private () at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:95
#1  0x00007fd5c50270f6 in __L_lock_740 () from /lib/x86_64-linux-gnu/libc.so.6
#2  0x00007fd5c5026f2a in __unsetenv (name=0x400b24 "bar") at setenv.c:325
#3  0x0000000000400a6d in main () at fork.c:41
```

可以看出调用unsetenv的时候，子进程就被卡住了。

为什么？

现在的库函数，为了做到可重入，其内部维护的变量通常会使用互斥量来保护。这些锁对用户一般是透明的，用户也不关心。setenv和unsetenv就是这样。尽管上述代码并没有显式地定义，但是进程内部已经维护了一个互斥量。

互斥量中维护了一个锁的值：0表示未上锁，1表示已上锁但是没有等待线程，2表示已上锁，并且有线程等待该锁。对于我们的例子而言，由于线程每100微秒就执行一次setenv，很有可能在主线程调用fork创建子进程的瞬间，互斥量的值是1。而这个值1被拷贝到了子进程。

对于父进程而言互斥量的值是1自然没有关系，因为父进程中有线程worker不停地加锁、解锁。但是子进程的情况就不同了，子进程中并没有worker。子进程自创建成功开始，setenv相关的互斥量的值就一直是1。子进程调用unsetenv函数时，“地雷”被引爆了。unsetenv无法获得互斥量，反而是通过调用futex系统调用陷入休眠，内核将其挂入对应的等待队列。

父进程的worker线程的解锁操作会唤醒子进程吗？

下面是内核get_futex_key函数中的部分代码：

```
if (!fshared) {
    if (unlikely(!access_ok(VERIFY_WRITE, uaddr, sizeof(u32))))
```

```
    return -EFAULT;
    key->private.mm = mm;
    key->private.address = address;
    get_futex_key_refs(key);
    return 0;
}
```

新建立的futex使用mm结构指针和地址address作为futex的键值，由于父子进程之间并不共享mm_struct，也就是说子进程的futex和父进程futex并不共享等待队列。换句话说，父进程通过setenv解锁时，根本就不会唤醒子进程。因此，子进程永远都不可能被唤醒了。

这仅仅是setenv/unsetenv函数，库函数中类似这种的函数并不少见：

- malloc函数的内部实现一定会有锁。

- printf系列的函数，其他线程可能持有stdout/stderr的锁。

- syslog函数内部实现也会用到锁。

综合上面的讨论，唯一安全的做法是，fork之后子进程立即调用exec执行另外的程序，彻底断绝子进程与父进程之间的关系，注意是立即，不要在调用exec之前执行任何语句，哪怕是不起眼的printf。

第9章 进程间通信：管道

在Linux系统中，有时候需要多个进程相互协作，共同完成某项任务。进程之间或线程之间有时候需要传递消息，有时候需要同步来协调彼此的工作。接下来的3章将讲述Linux中的进程间通信（interprocess communication，或者IPC）。

在第6章讲信号时曾提到，信号也是进程间通信的一种机制，尽管其主要作用不是这个。一个进程向另外一个进程发送信号，传递的信息是信号编号。当采用sigqueue函数发送信号时，还可以在信号上绑定数据（整型数字或指针），增强传递消息的能力。尽管如此，还是不建议将信号作为进程间通信的常规手段，原因在信号那一章中已经详细介绍过了。

在第7章讲线程时曾提到，线程在Linux中被实现为轻量级的进程，线程之间的同步手段（互斥量和条件等待），本质上也是进程间通信。

进程间通信的手段，大体可以分成以下两类：

第一类是通信类。这类手段的作用是在进程之间传递消息，交换数据。若细分下来，通信类也可以分成两种，一种是用来传递消息的（比如消息队列），另外一种是通过共享一片内存区域来完成信息的交换的（比如共享内存），如图9-1所示。

第二类是同步类。这类手段的目的是协调进程间的操作。某些操作，多个进程不能同时执行，否则可能会产生错误的结果，这就需要同步类的工具来协调。主要的同步类手段如图9-2所示。

从历史的角度来说，Linux下进程间通信的手段基本上是从Unix平台继承而来的。

管道是第一个广泛应用的进程间通信手段。日常在终端执行shell命令时，会大量用到管道。但管道的缺陷在于只能在有亲缘关系（有共同的祖先）的进程之间使用。为了突破这个限制，后来引入了命名管道。

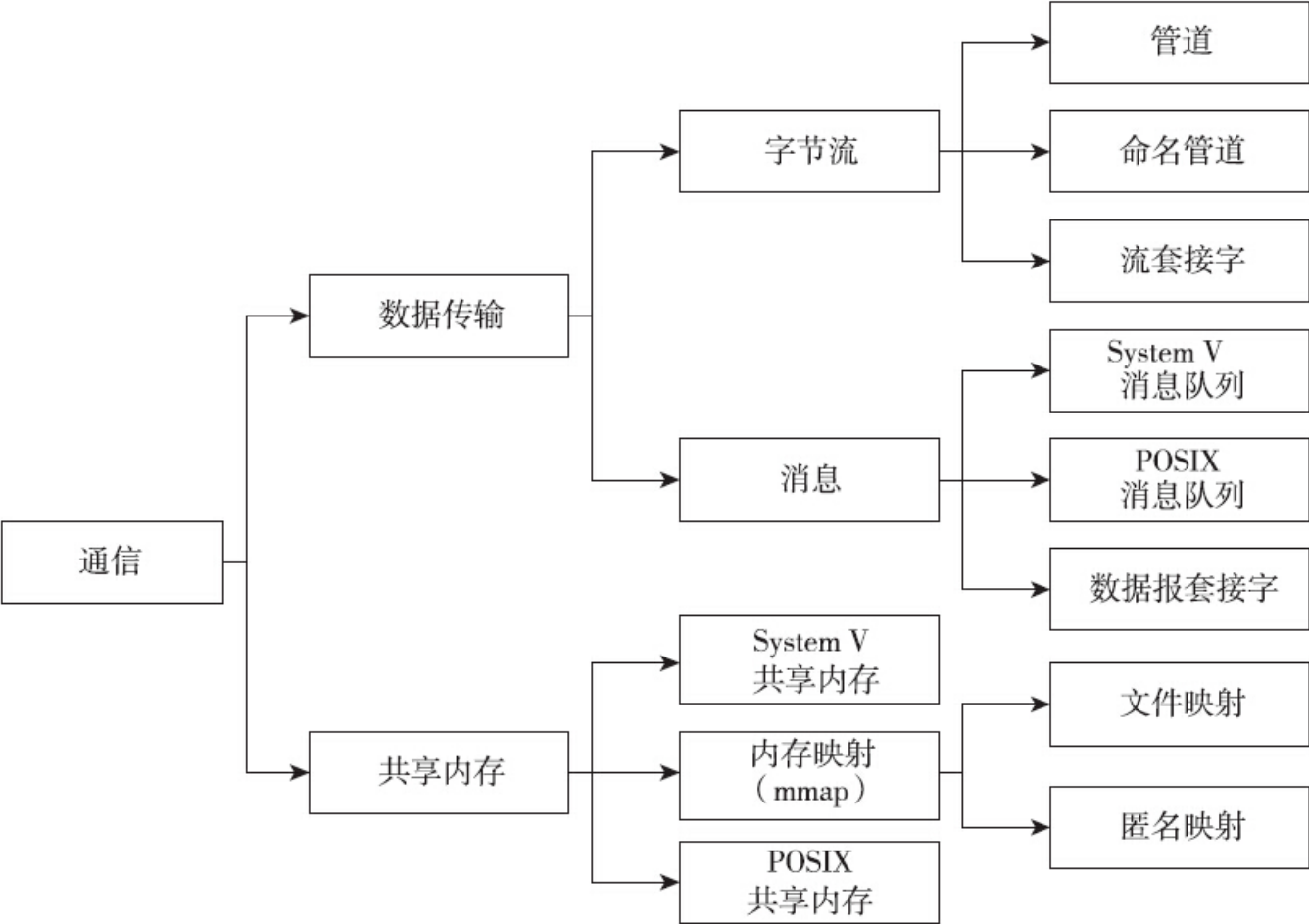


图9-1 通信类工具

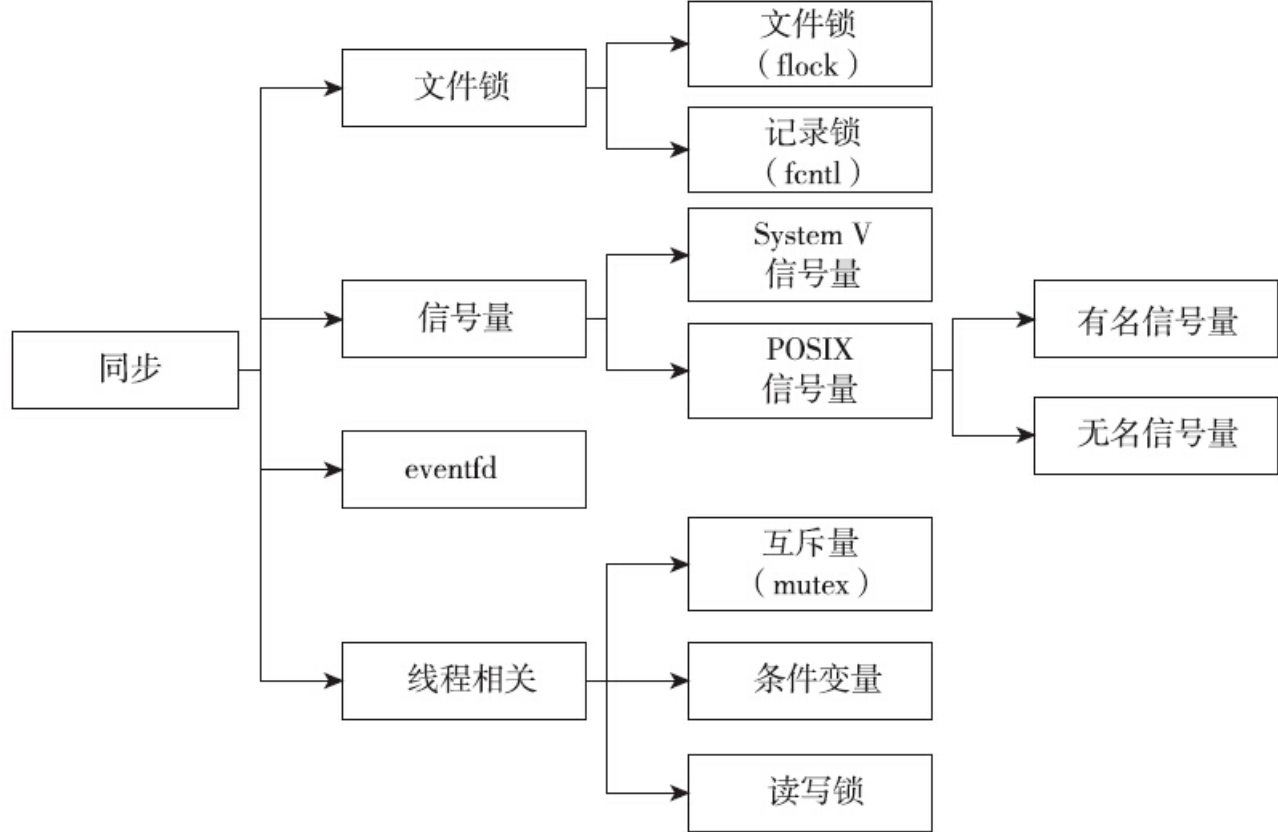


图9-2 同步类工具

接下来AT&T的贝尔实验室和加州大学伯克利分校的伯克利软件发布中心（BSD）分别开发出了风格迥异的进程间通信手段。前者通过对早期的进程间通信手段的改进和扩充，开发出System V IPC，包括消息队列、信号量和共享内存。但是这些方法，将进程间的通信始终局限在单个计算机这个范围之内。BSD则走了一条完全不同的道路，开发出了套接字（socket），跳出了单机的限制，可以实现不同计算机之间的进程间通信。Linux将System V IPC和BSD socket都继承了下来，丰富了进程间通信的方法。

System V IPC方法出现地比较早，几乎所有的Unix平台都支持System V IPC，其可移植性较好，但是在使用过程中也暴露出一些弱点。POSIX IPC提供了和System V IPC相对应的工具（它也包括消息队列、信号量和共享内存），它的出现晚于System V IPC。System V IPC广泛应用了一段时间后，才开始设计POSIX IPC的，因此，设计者可以借鉴System V IPC的长处，避免其缺点。从设计的角度上讲，POSIX IPC是优于System V IPC的，接口简单，易于使用。但是POSIX IPC的可移植性并不如System V IPC。

下面将分别介绍进程间通信的工具。其中的套接字在后面会有专门的章节来介绍，就不在进程间通信部分提及了。考虑到进程间通信的内容比较多，所以一共分成三章依次介绍，本章将主要介绍管道和命名管道。

9.1 管道

9.1.1 管道概述

管道是最早出现的进程间通信的手段。在shell中执行命令，经常会将上一个命令的输出作为下一个命令的输入，由多个命令配合完成一件事情。而这就是通过管道来实现的。

在图9-3中，进程who的标准输出，通过管道传递给下游的wc进程作为标准输入，从而通过相互配合完成了一件任务。

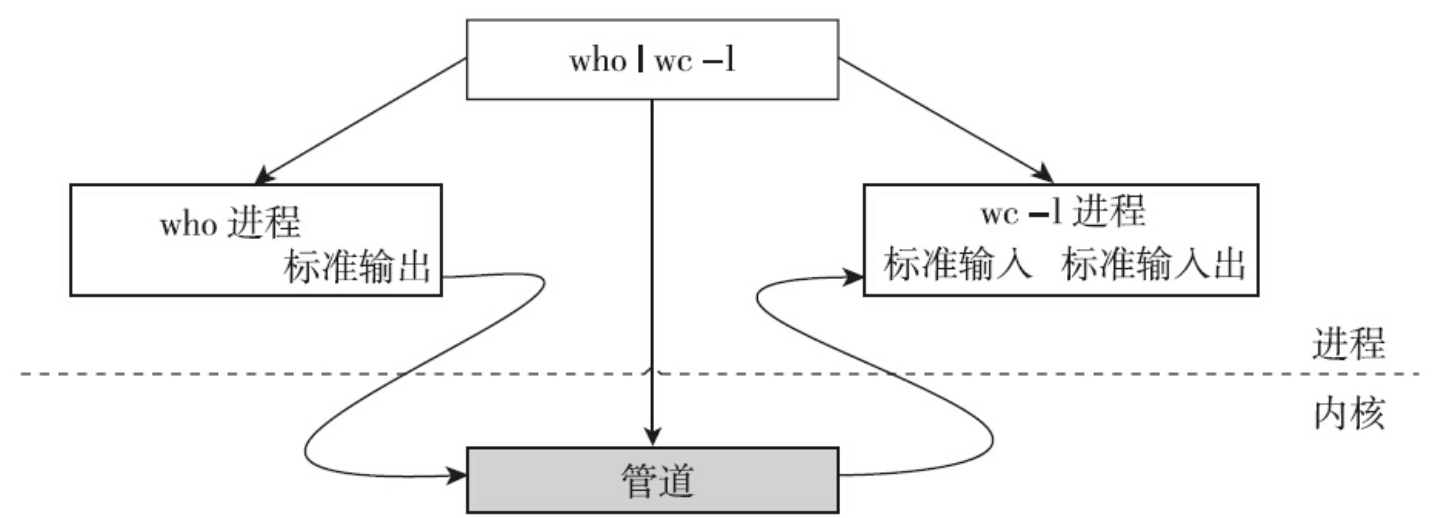


图9-3 管道的示意图

管道的作用是在有亲缘关系的进程之间传递消息。所谓有亲缘关系，是指有一个共同的祖先。所以管道并非只能用于父子进程之间，也可以用在兄弟进程之间，还可以用于祖孙进程之间甚至是叔侄进程之间。总而言之，只要共同的祖先曾经调用了pipe函数，打开的管道文件就会在fork之后，被各个后代进程所共享。打开的管道文件，就像是创建了一个家族私密场所，由远祖进程来创建，家族所有成员都知晓。家族成员可以将消息存放在该私密场所，等待另外一个接头的家族成员来取走消息，阅后即焚。

严格来说，家族里面的多个进程都可以往同一个秘密场所里面扔消息，也可以都从同一个秘密场所里面取消息，但是真的这么做的话又会存在风险。管道实质是一个字节流，并非前面提到的消息，没有消息的边界。如果多个进程发送的字节流混在一起，则无法辨认出各自的内容。所以一般是两个有亲缘关系的进程用管道来通信。从程序设计的角度来讲，当进程调用pipe函数时，哪两个有亲缘关系的进程使用该管道来通信应是事先约定好的，其他有亲缘关系的进程不应该进来搅局。其他进程想通信怎么办？那就创建它们之间需要用的另外的管道。

前面曾提到过，管道中的内容是阅后即焚的，这个特性指的是读取管道内容是消耗型的行为，即一个进程读取了管道内的一些内容之后，这些内容就不会继续在管道之中了。一般来讲管道是单向的。一个进

程负责往管道里面写内容，另外一个进程读取管道里的内容。若两个有亲缘关系的进程发扬二杆子精神，都要往管道里面写，都要往管道里面读，自然也是可以的，但是管道中的内容可能会变得混乱，从而无法完成通信的任务。如果两个进程之间想双向通信怎么办？可以建立两个管道，如图9-4所示。

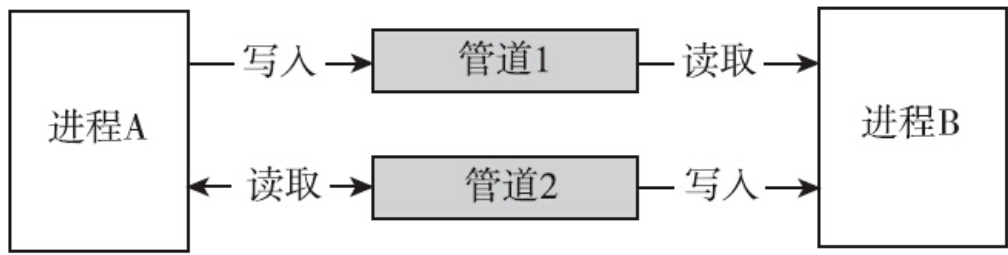


图9-4 利用两个管道双向通信

管道是一种文件，可以调用read、write和close等操作文件的接口来操作管道。另一方面管道又不是一种普通的文件，它属于一种独特的文件系统：pipefs。管道的本质是内核维护了一块缓冲区与管道文件相关联，对管道文件的操作，被内核转换成对这块缓冲区内存的操作。下面我们来看一下如何使用管道。

9.1.2 管道接口

在Linux下，可以使用如下接口创建管道：

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

如果成功，则返回值是0，如果失败，则返回值是-1，并且设置errno。需要处理的errno如表9-1所示。

表9-1 pipe函数的出错情况

errno	原 因
EMFILE	该进程使用的文件描述符已经多于 MAX_OPEN-2
ENFILE	系统中同时打开的文件已经超过了系统的限制
EFAULT	pipefd 参数不合法

成功调用pipe函数之后，会返回两个打开的文件描述符，一个是管道的读取端描述符pipefd[0]，另一个是管道的写入端描述符pipefd[1]。管道没有文件名与之关联，因此程序没有选择，只能通过文件描述符来访问管道，只有那些能看到这两个文件描述符的进程才能够使用管道。那么谁能看到进程打开的文件描述符呢？只有该进程及该进程的子孙进程才能看到。这就限制了管道的使用范围。

成功调用pipe函数之后，可以对写入端描述符pipefd[1]调用write，向管道里面写入数据，代码如下所示：

```
write(pipefd[1],wbuf,count);
```

一旦向管道的写入端写入数据后，就可以对读取端描述符pipefd[0]调用read，读出管道里面的内容。如下所示，管道上的read调用返回的字节数等于请求字节数和管道中当前存在的字节数的最小值。如果当前管道为空，那么read调用会阻塞（如果没有设置O_NONBLOCK标志位的话）。

```
read(pipefd[0],rbuf,count);
```

管道一端是写入端（pipefd[1]），另一端是读取端（pipefd[0]）。不应该对读取端描述符调用写操作，也不应该对写入端描述符调用读操作。如果我二杆子精神爆发，非要向读取端描述符写入，或者读取写入端描述符，结果会怎么样？

调用pipe函数返回的两个文件描述符中，读取端pipefd[0]支持的文件操作定义在read_pipefifo_fops，写入端pipefd[1]支持的文件操作定义在write_pipefifo_fops，其定义如下：

```
const struct file_operations read_pipefifo_fops = {
    .llseek      = no_llseek,
    .read        = do_sync_read,
    .aio_read    = pipe_read,
    .write       = bad_pipe_w,
    .poll        = pipe_poll,
    .unlocked_ioctl = pipe_ioctl,
    .open        = pipe_read_open,
    .release     = pipe_read_release,
    .fsync       = pipe_read_fsync,
};
const struct file_operations write_pipefifo_fops = {
    .llseek      = no_llseek,
    .read        = bad_pipe_r,
    .write       = do_sync_write,
    .aio_write   = pipe_write,
    .poll        = pipe_poll,
    .unlocked_ioctl = pipe_ioctl,
    .open        = pipe_write_open,
    .release     = pipe_write_release,
    .fsync       = pipe_write_fsync,
};
```

我们可以看到，对读取端描述符执行write操作，内核就会执行bad_pipe_w函数；对写入端描述符执行read操作，内核就会执行bad_pipe_r函数。这两个函数比较简单，都是直接返回-EBADF。因此对应的read和write调用都会失败，返回-1，并置errno为EBADF。

```
static ssize_t
bad_pipe_r(struct file *filp, char __user *buf, size_t count, loff_t *ppos)
{
    return -EBADF;
}
static ssize_t
bad_pipe_w(struct file *filp, const char __user *buf, size_t count, loff_t *ppos)
{
    return -EBADF;
}
```

我们只介绍了pipe函数接口，至今尚看不出来该如何使用pipe函数进行进程间通信。调用pipe之后，进程发生了什么呢？请看图9-5。

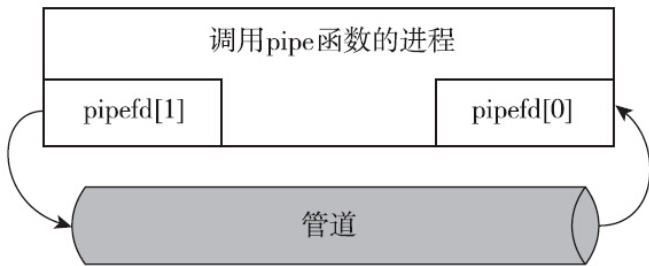


图9-5 进程调用pipe函数后

可以看到，调用pipe函数之后，系统给进程分配了两个文件描述符，即pipe函数返回的两个描述符。该进程既可以往写入端描述符写入信息，也可以从读取端描述符读出信息。可是一个进程管道，起不到任何通信的作用。这不是通信，而是自言自语。

如果调用pipe函数的进程随后调用fork函数，创建了子进程，情况就不一样了。fork以后，子进程复制了父进程打开的文件描述符（如图9-6所示），两条通信的通道就建立起来了。此时，可以是父进程往管道里写，子进程从管道里面读；也可以是子进程往管道里写，父进程从管道里面读。这两条通路都是可选的，但是不能都选。原因前面介绍过，管道里面是字节流，父子进程都写、都读，就会导致内容混在一起，对于读管道的一方，解析起来就比较困难。常规的使用方法是父子进程一方只能写入，另一方只能读出，管道变成一个单向的通道，以方便使用。如图9-7所示，父进程放弃读，子进程放弃写，变成父进程写入，子进程读出，成为一个通信的通道。

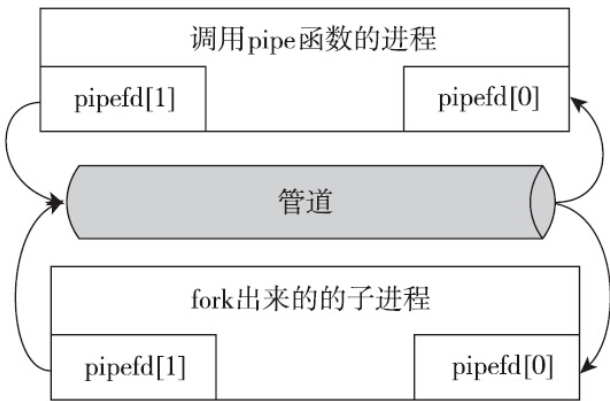


图9-6 fork之后

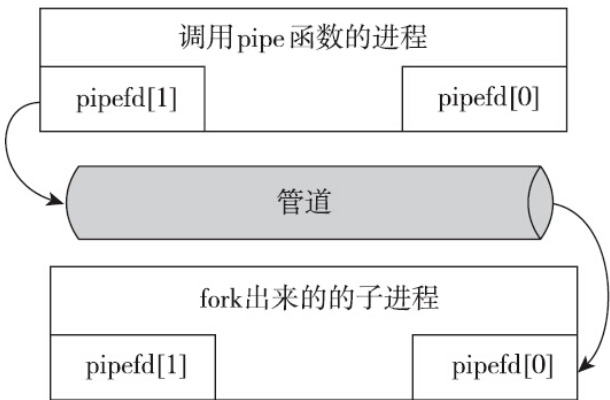


图9-7 fork之后，各自关闭不用的文件描述符

父进程如何放弃读，子进程又如何放弃写？其实很简单，父进程把读端口pipefd[0]这个文件描述符关闭掉，子进程把写端口pipefd[1]这个文件描述符关闭掉就可以了，示例代码如下：

```
int pipefd[2];
pipe(pipefd);
switch(fork())
{
case -1:
    /*fork failed, error handler here*/
```

```
case 0:      /*子进程

*/
    close(pipefd[1]) ; /*关闭掉写入端对应的文件描述符

*/
    /*子进程可以对

pipefd[0]调用

read*/
    break;

default: /*父进程

*/
    close(pipefd[0]); /*父进程关闭掉读取端对应的文件描述符

*/
    /*父进程可以对

pipefd[1]调用

write, 写入想告知子进程的内容

*/
    break
}
```

从内核的角度看，调用pipe之后，系统给进程分配了两个文件描述符，调用fork之后，子进程也就有了与管道对应的两个文件描述符。和普通文件不同，这两个文件描述符对应的是一块内存缓冲区域，如图9-8所示。

图9-8也讲述了如何在兄弟进程之间通过管道通信。如图9-8所示，父进程再次创建一个子进程B，子进程B就持有管道写入端，这时候两个子进程之间就可以通过管道通信了。父进程为了不干扰两个子进程通信，很自觉地关闭了自己的写入端。从此管道成为了两个子进程之间的单向的通信通道。在shell中执行管道命令就是这种情景，只是略有特殊之处，其特殊的地方是管道描述符占用了标准输入和标准输出两个文件描述符。

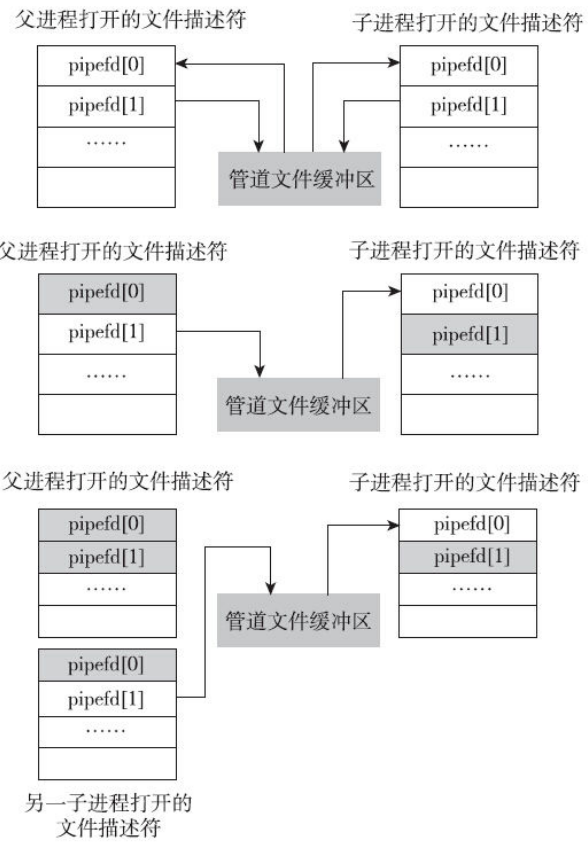


图9-8 有亲缘关系的进程通过管道来通信

9.1.3 关闭未使用的管道文件描述符

前面提到过，用管道通信的两个进程，各持有一个管道文件描述符，不相干的进程应自觉关闭掉这些文件描述符。这么做不仅仅是为了让数据的流向更加清晰，也不仅仅是为了节省文件描述符，更重要的原因是：关闭未使用的管道文件描述符对管道的正确使用影响重大。

管道有如下三条性质：

- 只有当所有的写入端描述符都已关闭，且管道中的数据都被读出，对读取端描述符调用read函数才会返回0（即读到EOF标志）。
- 如果所有读取端描述符都已关闭，此时进程再次往管道里面写入数据，写操作会失败，`errno`被设置为EPIPE，同时内核会向写入进程发送一个SIGPIPE的信号。
- 当所有的读取端和写入端都关闭后，管道才能被销毁。

由于管道具有这些特性，因此我们要及时关闭没用的管道文件描述符，下面我们来细细分析这样做的原因。

1.关闭无用的管道写入端

从管道读取数据的进程，须要关闭其持有的管道写入端描述符。不参与通信的其他有亲缘关系的进程也应该关闭管道写入端描述符。

管道也符合生产者-消费者模型。写入管道，对应于生产内容；读取管道，对应于消费内容。当所有的生产者都退场以后，消费者应有办法判断这种情况，而不是傻傻地等待已不复存在的生产者继续生产内容，以至于陷入永久的阻塞。

如何判断？

答案是通过文件结束标志EOF。当对管道读取端调用read函数返回0时，就意味着所有的生产者都退场了，作为消费者的读取进程，就不需要再继续等待新的内容了。

什么情况下对管道读取端描述符调用read会返回0呢？

- 所有相关的进程都已经关闭了管道的写入端描述符。
- 管道中已有内容都被读取完毕。

同时满足上述条件，对管道读取端调用read会返回0。根据这个消费者就可以判断管道内容的生产者已经不存在了，它也不必傻傻等待，可以关闭读取端描述符了。

从上面的讨论可以看出，如果负责读取的进程，或者与通信无关的进程，不关闭管道的写入端描述符，就会有管道写入端描述符泄漏。当所有负责写入的进程都关闭了写入端描述符后，负责读的进程调用read时，仍会阻塞于此（如果没有设置O_NONBLOCK标志位的话），而且永不返回。这是因为内核维护的引用计数发现还有进程可以写入管道，因此read函数依旧会阻塞。

这个流程可以通过一个例子来验证。

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int pipe_fd[2];
    pid_t pid;
    char r_buf[4096];
    char w_buf[4096];
    int writenum;
    int rnum;
    memset(r_buf,0,sizeof(r_buf));
    if(pipe(pipe_fd)<0)
    {
        printf("[PARENT] pipe create error\n");
        return -1;
    }
    if((pid=fork()) == 0)
    {
        /*如果孩子进程忘记关闭管道写入端，那么，即使父进程关闭了写入端，

while循环也无法跳出

*/
        close(pipe_fd[1]);

        while(1)
        {
            rnum = read(pipe_fd[0],r_buf,1000);
            printf("[CHILD ] readnum is %d\n",rnum);
            if(rnum == 0) /*meet EOF*/
            {
                printf("[CHILD ] all the writer of pipe are closed. break and exit.\n");
                break;
            }
        }
        close(pipe_fd[0]);
        exit(0);
    }
    else if(pid>0)
    {
        close(pipe_fd[0]);
        memset(w_buf,0,sizeof(w_buf));
        if((writenum = write(pipe_fd[1],w_buf,1024)) == -1)
            printf("[PARENT] write to pipe error\n");
        else
        {
            printf("[PARENT] the bytes write to pipe is %d \n", writenum);
        }
        sleep(15);
        printf("[PARENT] I will close the last write end of pipe.\n");
        close(pipe_fd[1]);
        sleep(2);
        return 0;
    }
}
```

在上面的例子中，父子进程通过管道进行通信，父进程关闭了管道的读取端，子进程关闭了管道的写入端。父进程写入了1024字节，子进程则在循环体中调用`read`，每次尝试读取1000字节。子进程很快就读完了父进程生产的1024字节。但是父进程并没有立刻关闭管道的写入端，而是睡眠了15秒后，才关闭管道写入端。从子进程读完父进程生产的1024字节开始，到父进程关闭管道写入端这段接近15秒的时间内，子进程实际上是阻塞在`read`函数上的。当父进程关闭管道写入端，子进程调用的`read`函数才得以返回，返回值是0。子进程看到返回值0后，意识到硕果仅存的管道写入端也不复存在了，所以它没必要再继续`read`了，于是子进程就跳出了循环体。

示例代码的输出如下，与我们上面分析的一样：

```
[PARENT] the bytes write to pipe is 1024
[CHILD ] readnum is 1000
[CHILD ] readnum is 24
[PARENT] I will close the last write end of pipe.
[CHILD ] readnum is 0
[CHILD ] all the writer of pipe are closed. break and exit
```

父子进程配合地珠联璧合，但是如果子进程忘记关闭管道的写入端，（删除上面示例代码中加粗的一行）结局就大相径庭了。纵然父进程关闭了管道的写入端，但是因为管道仍然存在一个写入端，所以子进程的`read`函数依旧会阻塞，无法返回。这显然不是我们期待的结果。

2.关闭无用的管道读取端

如果对管道的写入端描述符调用`write`函数，则会走到内核的`pipe_write`函数。在该函数中可以看到如下代码：

```
if (!pipe->readers) {
    send_sig(SIGPIPE, current, 0);
    ret = -EPIPE;
    goto out;
}
```

当管道的读取端不复存在时，内核会向`write`函数的调用进程发送SIGPIPE信号，并且当前的`write`系统调用失败，错误码为EPIPE。

SIGPIPE信号默认情况下会杀死一个进程，当然我们也可以捕获或忽略该信号。事实上大多数情况下，服务器端的程序都会将SIGPIPE的信号处理函数设置成SIG_IGN，忽略掉该信号。这样的话，`write`系统调用就会返回失败，`errno`是EPIPE，通过返回值和`errno`，就可以及时获知所有的读取端都已关闭了。

当所有的管道读取端都不复存在时，管道的写入操作就会失败。为何要如此设计？

因为管道的读取端是管道内容的消费者，管道的写入端是管道内容的生产者。当消费者已经不复存在了，生产者自然没有继续生产的必要了。对于这个道理，电视剧《亮剑》中的山本一木都很清楚：

没有了观众，也就没有了表演。

所以不参与通信的进程，以及负责向管道写入内容的进程应该及时地关闭管道的读取端描述符。只有这样，当通信双方中的消费者关闭管道读取端时，管道内容的生产者才能在第一时间获知所有消费者都已不存在了这个事实。

如果写入管道的进程不关闭管道的读取文件描述符，哪怕其他进程都已经关闭了读取端，该进程仍可以向管道写入数据，但是只有生产者，没有消费者，管道最终会被写满，当管道被写满后，后续的写入请求就会被阻塞。

下面通过实例来证实：当最后一个读取端关闭时，向管道写入会触发SIGPIPE信号，同时write会返回失败，`errno`为EPIPE。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
void sighandler(int signo);
int main(void)
{
    int fds[2];
    if(signal(SIGPIPE,sighandler) == SIG_ERR)
    {
        fprintf(stderr,"signal error (%s)\n",strerror(errno));
        exit(EXIT_FAILURE);
    }
    if(pipe(fds) == -1)
    {
        fprintf(stderr,"create pipe failed(%s)\n",strerror(errno));
        exit(EXIT_FAILURE);
    }
    pid_t pid;
    pid = fork();
    if(pid == -1)
    {
        fprintf(stderr,"fork error (%s)\n",strerror(errno));
        exit(EXIT_FAILURE);
    }
    if(pid == 0)
    {
        fprintf(stdout,"[CHILD ] I will close the last read end of pipe \n");
        close(fds[0]); //子进程关闭读取端文件描述符

        exit(EXIT_SUCCESS);
    }
    close(fds[0]); //父进程关闭读取端文件描述符

    sleep(1); //确保子进程也将读取端关闭
```

```

int ret;
ret = write(fds[1], "hello", 5);
if (ret == -1)
{
    fprintf(stderr, "[PARENT] write error(%s)\n", strerror(errno));
}
return 0;
}
void sighandler(int signo)
{
    printf("[PARENT] catch a SIGPIPE signal and signum = %d\n", signo);
}

```

fork之后，父子进程都立刻关闭了读取端，这时候，管道已经不存在任何读取端了。1秒钟之后，父进程尝试向管道写入。此时按照前面的分析，父进程应该会收到SIGPIPE信号，write返回失败，并且errno为EPIPE。父进程为SIGPIPE安装了信号处理函数，如果收到SIGPIPE信号，会有打印提示。下面来看看程序的输出：

```

[CHILD ] I will close the last read end of pipe
[PARENT] catch a SIGPIPE signal and signum = 13
[PARENT] write error(Broken pipe)

```

通过上面的讨论可以看出，正常使用管道的场景，应该只有两个进程和管道关联，一个进程只拥有管道的写入端，另一个进程只拥有管道的读取端。

如何检验管道是否满足上面的情形？以如下情况为例：

```

int pipefd[2]
ret = pipe(pipefd);
fork()

```

管道是文件的一种，在/proc/PID/fd/下可以看到打开的管道文件，如下所示：

```

manu@manu-rush:~$ ll /proc/2889/fd
total 0
dr-x----- 2 manu manu  0 Jul 24 00:13 ./
dr-xr-xr-x  9 manu manu  0 Jul 24 00:13 ../
lrwx----- 1 manu manu 64 Jul 24 00:13 0 -> /dev/pts/0
lrwx----- 1 manu manu 64 Jul 24 00:13 1 -> /dev/pts/0
lrwx----- 1 manu manu 64 Jul 24 00:13 2 -> /dev/pts/0
lr-x----- 1 manu manu 64 Jul 24 00:13 3 -> pipe:[13870]
l-wx----- 1 manu manu 64 Jul 24 00:13 4 -> pipe:[13870]

```

可以看出文件描述符3和4都是管道文件，其后面的相同数字13870表示它们属于同一个管道。文件描述符3对应的文件属性中有r，表示管道的读取端，文件描述符4对应的文件属性中有w表示4是管道的写入端。

还有哪些进程持有管道对应的文件描述符？

```

manu@manu-rush:~$ lsof | grep FIFO | grep 13870
pipe      2889      manu    3r      FIFO    0,8        0t0    13870  pipe
pipe      2889      manu    4w      FIFO    0,8        0t0    13870  pipe
pipe      2890      manu    3r      FIFO    0,8        0t0    13870  pipe
pipe      2890      manu    4w      FIFO    0,8        0t0    13870  pipe

```

从上面的输出可以知晓，管道13870并不满足前面的讨论。在理想情况下，输出应该只有两行，一个进程只有管道的写入端，另一个进程只有管道的读取端。

9.1.4 管道对应的内存区大小

管道本质是一片内存区域，自然有大小。自Linux 2.6.11版本起，管道的默认大小是65536字节，可以调用fcntl来获取和修改这个值的大小，代码如下：

获取管道的大小

```
pipe_capacity = fcntl(fd, F_GETPIPE_SZ); 设置管道的大小
```

```
ret = fcntl(fd, F_SETPIPE_SZ, size);
```

管道内存区域的大小必须在页面大小（PAGE）和上限值之间，其上限记录在/proc/sys/fs/pipe-max-size里，对于特权用户，还可以修改该上限值。

```
cat /proc/sys/fs/pipe-max-size
1048576
```

管道的容量可以扩大，自然也可以缩小。缩小管道容量时会遇到一种比较有意思的场景，即当前管道中已存在的内容大于fcntl函数调用中指定的size，此时fcntl函数会返回失败，并置错误码为EBUSY。

管道容量有大小这个事实对于编程有什么影响呢？

在使用管道的过程中要意识到：管道有大小，写入须谨慎，不能连续地写入大量的内容，一旦管道满了，写入就会被阻塞；对于读取端，要及时地读取，防止管道被写满，造成写入阻塞。

9.1.5 shell管道的实现

shell编程会大量使用管道，我们经常看到前一个命令的标准输出作为后一个命令的标准输入，来协作完成任务，如图9-9所示。管道是如何做到的呢？

兄弟进程可以通过管道来传递消息，这并不稀奇，前面已经图示了做法。关键是如何使得一个程序的标准输出被重定向到管道中，而另一个程序的标准输入从管道中读取呢？



图9-9 管道在shell中的应用

答案就是复制文件描述符。

对于第一个子进程，执行dup2之后，标准输出对应的文件描述符1，也成为了管道的写入端。这时候，管道就有了两个写入端，按照前面的建议，需要关闭不相干的写入端，使读取端可以顺利地读到EOF，所以应将刚开始分配的管道写入端的文件描述符pipefd[1]关闭掉。

```
if(pipefd[1] != STDOUT_FILENO)
{
    dup2(pipefd[1],STDOUT_FILENO);
    close(pipefd[1]);
}
```

同样的道理，对于第二个子进程，如法炮制：

```
if(pipefd[0] != STDIN_FILENO)
{
    dup2(pipefd[0],STDIN_FILENO);
    close(pipefd[0]);
}
```

简单来说，就是第一个子进程的标准输出被绑定到了管道的写入端，于是第一个命令的输出，写入了管道，而第二个子进程管道将其标准输入绑定到管道的读取端，只要管道里面有了内容，这些内容就成了标准输入。

两个示例代码，为什么要判断管道的文件描述符是否等于标准输入和标准输出呢？原因是，在调用pipe时，进程很可能已经关闭了标准输入和标准输出，调用pipe函数时，内核会分配最小的文件描述符，所以pipe的文件描述符可能等于0或1。在这种情况下，如果没有if判断加以保护，代码就变成了：

```
dup2(1,1);
close(1);
```

这样的话，第一行代码什么也没做，第二行代码就把管道的写入端给关闭了，于是便无法传递信息了。

9.1.6 与shell命令进行通信 (popen)

管道的一个重要作用是和外部命令进行通信。在日常编程中，经常会需要调用一个外部命令，并且要获取命令的输出。而有些时候，需要给外部命令提供一些内容，让外部命令处理这些输入。Linux提供了popen接口来帮助程序员做这些事情。

就像system函数，即使没有system函数，我们通过fork、exec及wait家族函数一样也可以实现system的功能。但终究是不方便，system函数为我们提供了一些便利。同样的道理，只用pipe函数及dup2等函数，也能完成popen要完成的工作，但popen接口给我们提供了便利。

popen接口定义如下：

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

popen函数会创建一个管道，并且创建一个子进程来执行shell，shell会创建一个子进程来执行command。根据type值的不同，分成以下两种情况。

如果type是r：command执行的标准输出，就会写入管道，从而被调用popen的进程读到。通过对popen返回的FILE类型指针执行read或fgets等操作，就可以读取到command的标准输出，如图9-10所示。

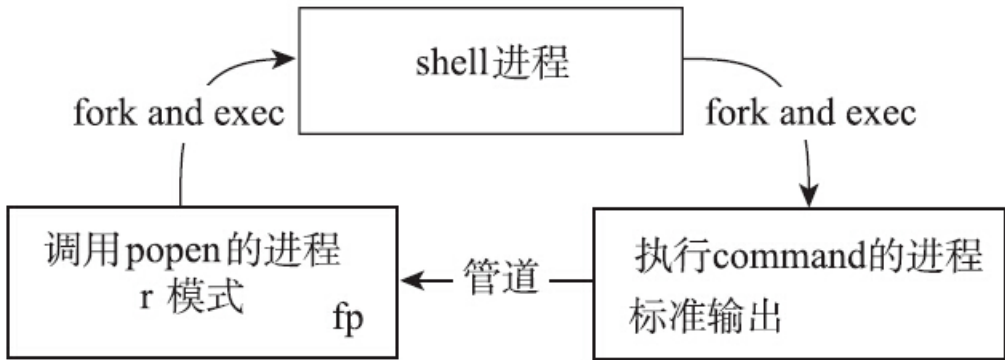


图9-10 r模式调用popen

如果type是w：调用popen的进程，可以通过对FILE类型的指针fp执行write、fputs等操作，负责往管道里面写入，写入的内容经过管道传给执行command的进程，作为命令的输入，如图9-11所示。

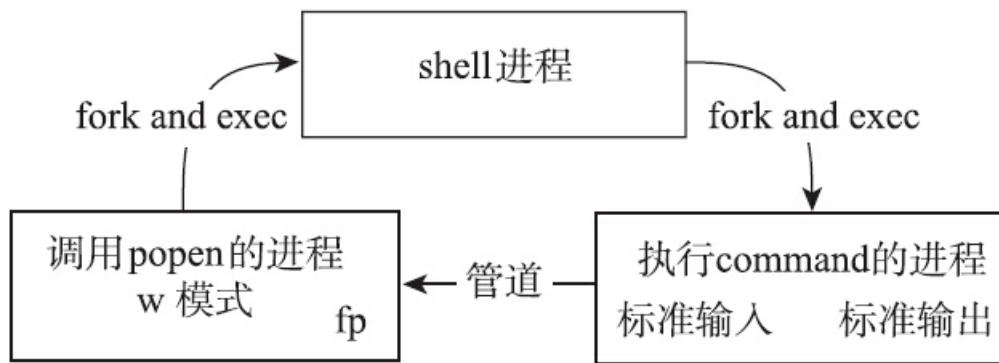


图9-11 w模式调用popen

popen函数成功时，会返回stdio库封装的FILE类型的指针，失败时会返回NULL，并且设置errno。常见的失败有fork失败，pipe失败，或者分配内存失败。

I/O结束了以后，可以调用pclose函数来关闭管道，并且等待子进程的退出。尽管popen函数返回的是FILE类型的指针，也不应调用fclose函数来关闭popen函数打开的文件流指针，因为fclose不会等待子进程的退出。pclose函数成功时会返回子进程中shell的终止状态。popen函数和system函数类似，如果command对应的命令无法执行，就如同执行了exit(127)一样。如果发生其他错误，pclose函数则返回-1。可以从errno中获取到失败的原因。

下面给出一个简单的例子，来示范下popen的用法：

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<errno.h>
#include<sys/wait.h>
#include<signal.h>
#define MAX_LINE_SIZE 8192
void print_wait_exit(int status)
{
    printf("status = %d\n",status);
    if(WIFEXITED(status))
    {
        printf("normal termination,exit status = %d\n",WEXITSTATUS(status));
    }
    else if(WIFSIGNALED(status))
    {
        printf("abnormal termination,signal number =%d%s\n",
            WTERMSIG(status),
#ifdef WCOREDUMP
            WCOREDUMP(status)?"core file generated" : "");
#else
            "");
#endif
    }
}
int main(int argc ,char* argv[])
{
    FILE *fp = NULL ;
    char command[MAX_LINE_SIZE],buffer[MAX_LINE_SIZE];
    if(argc != 2 )
    {
        fprintf(stderr,"Usage: %s filename \n",argv[0]);
        exit(1);
    }
    snprintf(command,sizeof(command),"cat %s",argv[1]);
    fp = popen(command,"r");
    if(fp == NULL)
    {
        fprintf(stderr,"popen failed (%s)",strerror(errno));
        exit(2);
    }
}

```

```

while(fgets(buffer,MAX_LINE_SIZE,fp) != NULL)
{
    fprintf(stdout,"%s",buffer);
}
int ret = pclose(fp);
if(ret == 127 )
{
    fprintf(stderr,"bad command : %s\n",command);
    exit(3);
}
else if(ret == -1)
{
    fprintf(stderr,"failed to get child status (%s)\n",
strerror(errno));
    exit(4);
}
else
{
    print_wait_exit(ret);
}
exit(0);
}

```

将文件名作为参数传递给程序，执行`cat filename`的命令。`popen`创建子进程来负责执行`cat filename`的命令，子进程的标准输出通过管道传给父进程，父进程可以通过`fgets`来读取`command`的标准输出。

`popen`函数和`system`有很多相似的地方，但是也有显著的不同。调用`system`函数时，`shell`命令的执行被封装在了函数内部，所以若`system`函数不返回，调用`system`的进程就不再继续执行。但是`popen`函数不同，一旦调用`popen`函数，调用进程和执行`command`的进程便处于并行状态。然后`pclose`函数才会关闭管道，等待执行`command`的进程退出。换句话说，在`popen`之后，`pclose`之前，调用`popen`的进程和执行`command`的进程是并行的，这种差异带来了两种显著的不同：

- 在并行期间，调用`popen`的进程可能会创建其他子进程，所以标准规定`popen`不能阻塞`SIGCHLD`信号。这也意味着，`popen`创建的子进程可能被提前执行的等待操作所捕获。若发生这种情况，调用`pclose`函数时，已经无法等待`command`子进程的退出，这种情况下，将返回-1，并且`errno`为`ECHILD`。

- 调用进程和`command`子进程是并行的，所以标准要求`popen`不能忽略`SIGINT`和`SIGQUIT`信号。如果是从键盘产生的上述信号，那么，调用进程和`command`子进程都会收到信号。

9.2 命名管道FIFO

上一节介绍的管道也被称为无名管道。这种管道因为没有实体文件与之关联，靠的是世代相传的文件描述符，所以只能应用在有共同祖先的各个进程之间。对于没有亲缘关系的任意两个进程之间，无名管道就爱莫能助了。

命名管道就是为了解决无名管道的这个问题而引入的。FIFO与管道类似，最大的差别就是有实体文件与之关联。由于存在实体文件，不相关的没有亲缘关系的进程也可以通过使用FIFO来实现进程之间的通信。

与无名管道相比，命名管道仅仅是披了一件马甲，其核心与无名管道是一模一样的。内核的fs/fifo.c文件仅有153行，说白了，这简短的代码只干了两件事：

- 从外表看，我是一个FIFO文件，有文件名，任何进程通过文件名都可以打开我。
- 我的内心与无名管道是一样的，支持的文件操作与无名管道也是一样的。

9.2.1 创建FIFO文件

创建命名管道的接口定义如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

其中，第二个参数的含义是FIFO文件的读写执行权利，和open函数类似。当然真实的读写执行权限，还需要按照当前进程的umask来取掩码，即：

```
real_mode = (mode & ~umask)
```

除了用C接口，还可以用命令来创建一个命名管道：

```
mkfifo [-m mode] pathname
```

pathname是创建命名管道文件的文件名，-m mode的使用方法和chmod的方法一样。

除此外，mknod命令也可以用来创建FIFO文件，使用方法如下：

```
mknod [-m mode] pathname p
```

命令末尾的p表示要创建命名管道（named pipe）。

创建出来的FIFO文件，用ls-l来查看，第一个字母是p，表示这是命名管道文件。

```
prw-rw-r--  1 manu manu    0  2月
```

```
19 23:03 myfifo2
```

在shell编程中可以使用-p file来判断是否为FIFO文件。在C语言中如何判断是否为FIFO文件呢？通过S_ISFIFO宏可以判断，不过要先通过stat或fstat函数来获取到文件的属性信息，如下面的代码所示：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
S_ISFIFO(buf->st_mode)
```

9.2.2 打开FIFO文件

一旦FIFO文件创建好了，就可以把它用于进程间的通信了。一般的文件操作函数如open、read、write、close、unlink等都可以用在FIFO文件上。

FIFO文件和普通文件相比，有一个明显的不同：程序不应该以O_RDWR模式打开FIFO文件。POSIX标准规定，以O_RDWR模式打开FIFO文件，结果是未定义的。当然了，Linux提供了对O_RDWR的支持，在某些场景下，O_RDWR模式的打开是有价值的（9.4节给出了一个例子）。

对FIFO文件推荐的使用方法是，两个进程一个以只读模式（O_RDONLY）打开FIFO文件，另一个以只写模式（O_WRONLY）打开FIFO文件。这样负责写入的进程写入FIFO的内容就可以被负责读取的进程读到，从而达到通信的目的。

打开一个FIFO文件和打开普通文件相比，又有不同。在没有进程以写模式（O_RDWR或O_WRONLY）打开FIFO文件的情况下，以O_RDONLY模式打开一个FIFO文件时，调用进程会陷入阻塞，直到另一进程以O_WRONLY（或者O_RDWR）的标志位打开该FIFO文件为止。同样的道理，在没有进程以读模式（O_RDONLY或O_RDWR）打开FIFO文件的情况下，如果一个进程以O_WRONLY的标志位打开一个FIFO文件，调用进程也会阻塞，直到另一个进程以O_RDONLY（或者O_RDWR）的标志位打开该FIFO文件为止。也就是说，打开FIFO文件会同步读取进程和写入进程。

乍看之下，O_RDONLY模式打开不能返回，在等写打开，同样O_WRONLY打开不能返回，在等读打开，造成死锁，谁都返回不了。事实上不是这样的。当O_RDONLY打开和O_WRONLY打开的请求都到达FIFO文件时，两者就都能返回了。

内核之中，维护有引用计数r_counter和w_counter，分别记录FIFO文件两种打开模式的引用计数。对于FIFO文件，无论是读打开还是写打开，都会根据引用计数判断对方是否存在，进而决定后续的行为（是阻塞、返回成功，还是返回失败）。

FIFO文件提供了O_NONBLOCK标志位，该标志位会显著影响open的行为模式。将O_RDONLY、O_WRONLY及O_NONBLOCK三种标志位结合在一起考虑，共有以下四种组合方式，如表9-2所示。

表9-2 打开FIFO文件的不同情况

打开标志位	行 为 模 式
O_RDONLY	当已存在写打开该 FIFO 文件的进程时，成功返回
	当不存在写打开该 FIFO 文件的进程时，会陷入阻塞，直到有进程以 O_WRONLY 模式（或者 O_RDWR 模式）打开该 FIFO 文件，方能返回
O_RDONLY+O_NONBLOCK	当已存在写打开该 FIFO 文件的进程时，成功返回
	当不存在写打开该 FIFO 文件的进程时，亦成功返回
O_WRONLY	当已存在读打开该 FIFO 文件的进程时，成功返回
	当不存在读打开该 FIFO 文件的进程时，会陷入阻塞，直到有进程以 O_RDONLY 模式（或者 O_RDWR 模式）打开该 FIFO 文件，方能返回
O_WRONLY+O_NONBLOCK	当已存在读打开该 FIFO 文件的进程时，成功返回
	当不存在读打开该 FIFO 文件的进程时，返回 -1，并置 errno 为 ENXIO

同样是带O_NONBLOCK标志位的打开，没有写打开进程时，读打开请求可以成功返回，但没有读打开进程时，写打开请求却失败，返回-1，并置errno为ENXIO，两相比较，是否太不公平了？

这样设计是有原因的：FIFO只有读取端，没有写入端，并无显著的危害，所有尝试从FIFO中读取数据的操作都不会返回任何数据。反之则不然。如果允许只存在写入端，不存在读取端，那么open之后，所有向FIFO文件的写入操作，都会导致SIGPIPE信号的产生，以及write调用返回EPIPE的错误，所以在源头上堵住（即让open函数返回失败）反倒更加合理。

打开FIFO文件的内核代码位于内核的fs/fifo.c文件中，代码简短，非常易懂。读者可以通过阅读源代码，加深对打开FIFO文件的理解。

9.3 读写管道文件

无名管道pipe和命名管道FIFO在内核实现部分有很大的重叠，都属于管道文件系统（pipefs）。无名管道，分裂成了读取文件描述符和写入文件描述符。而命名管道则将两个描述符合二为一，如果是读打开，就如同获取到了无名管道的读取文件描述符；如果是写打开，就如同获取到了无名管道的写入文件描述符。这种本质上的一致，造成FIFO的读写控制和无名管道的读写控制是一模一样的，因此在本节一并介绍。

影响管道或FIFO文件读写行为的因素有：

- 当前管道中存在的字节数p。
- 是否有O_NONBLOCK标志位。
- 管道的最大容量PIPE_BUF和要读写的字节数n的关系。
- 读写端是否都存在。

管道文件的读写中一个很重要的标志位是O_NONBLOCK，该标志位会影响读写的行为模式。

对于无名管道，Linux提供了特有的pipe2函数，该函数的接口如下：

```
#define _GNU_SOURCE
#include <unistd.h>
int pipe2(int pipefd[2], int flags);
```

可选的flag就有O_NONBLOCK。

对于命名管道FIFO，打开文件时，可以带上O_NONBLOCK标志位来控制读写的行为（当然了，对于FIFO文件，O_NONBLOCK也会影响打开的行为）。

如果打开时，忘记带上O_NONBLOCK标志位，那该如何补救呢？答案是用fcntl这把文件控制的瑞士军刀。

通过如下代码，可以给管道文件加上O_NONBLOCK标志位：

```
int flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

相反的，如果打开时，带有O_NONBLOCK标志位，而后面又想取消该标志位，又该怎么做？

```
int flags = fcntl(fd, F_GETFL);
flags &= ~O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

花开两朵，各表一枝。先来说说从FIFO或管道读取端读，如表9-3所示。

表9-3 从一个包含p字节的管道或FIFO读取n字节的含义

	$p = 0$ 且 存在写入端描述符尚 未关闭	$p = 0$ 且 所有写入端描述符均 已关闭	$p < n$	$p \geq n$
未启用 O_NONBLOCK	阻塞	返回 0（EOF）	读取 p 字节	读取 n 字节
启用 O_NONBLOCK	失败（EAGAIN）	返回 0（EOF）	读取 p 字节	读取 n 字节

从表9-3可以看出：

- O_NONBLOCK标志位影响的仅仅是当管道为空并且存在写入端时的行为，读取操作的行为是阻塞，还是当即返回失败。
- 当read返回0时，表示已经遇到了EOF，并且所有的写入端都已经关闭了。这一般出现在管道的使命结束时，此时读取端也可以关闭了。

说完读，然后说写（如表9-4所示）。对于管道的写入而言，POSIX标准规定，如果一次写入的数据量不超过PIPE_BUF个字节，必须确保写

入是原子的（atomic）。所谓原子是指：写入的内容必须确保是连续的，纵然有多个进程同时往管道中写入，写入的内容也不会被其他进程写入的内容打断，本次写入的内容不会混杂其他进程write函数写入的内容。标准规定，PIPE_BUF最少为512字节，对于Linux而言，这个值是4096，一个页面的大小。

表9-4 向管道写入n字节

	无 NON_BLOCK 标志位	有 NON_BLOCK 标志位
写入字节数 $n \leq \text{PIPE_BUF}$ (保证写入的原子性)	当空闲区域不足以容纳 n 字节时，陷入阻塞，等待读取进程取走管道的部分内容	当管道空闲区域不足以容纳 n 字节时，立即返回失败，并置 <code>errno</code> 为 <code>EAGAIN</code>
写入字节数 $n > \text{PIPE_BUF}$ (不保证写入的原子性)	使命必达的策略。当空闲区域不足以容纳 n 字节时，陷入阻塞，待管道空间足够时再写入。但成功返回时，写入字节一定是 n	尽力而为的策略。当写满管道时，返回，实际写入字节数在 $1 \sim n$ 之间。用户需要判断返回值，来确定写入的字节数

关于单次写入的长度超出PIPE_BUF，内核不能保证其原子性这个事实，我们可以通过一个简单的实验来验证，示例代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#define BUF_4K 4*1024
#define BUF_8K 8*1024
#define BUF_12K 12*1024
int main(void)
{
    char a[BUF_4K];
    char b[BUF_8K];
    char c[BUF_12K];
    memset(a, 'A', sizeof(a));
    memset(b, 'B', sizeof(b));
    memset(c, 'C', sizeof(c));
    int pipefd[2];
    int ret = pipe(pipefd);
    if (ret == -1)
    {
        fprintf(stderr, "failed to create pipe (%s)\n", strerror(errno));
        return 1;
    }
    pid_t pid;
    pid = fork();
    if (pid == 0) // 第一个子进程

    {
        close(pipefd[0]);
        int loop = 0 ;
        while(loop++ < 10)
        {
            ret = write(pipefd[1], a, sizeof(a));
            printf("apid=%d write %d bytes to pipe\n", getpid(), ret);
        }
        exit(0);
    }
    pid = fork();
    if (pid == 0) // 第二个子进程

    {
        close(pipefd[0]);
        int loop = 0;
        while(loop++ < 10)
        {
            ret = write(pipefd[1], b, sizeof(b));
            printf("bpid=%d write %d bytes to pipe\n", getpid(), ret);
        }
        exit(0);
    }
    pid = fork();
    if (pid == 0) // 第三个子进程

    {
        close(pipefd[0]);
        int loop = 0;
        while(loop++ <10)
        {
            ret = write(pipefd[1], c, sizeof(c));
            printf("cpid=%d write %d bytes to pipe\n", getpid(), ret);
        }
        exit(0);
    }
    close(pipefd[1]);
    sleep(1);
    int fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    char buf[1024*4] = {0};
    int n = 1;
    while (1)
    {
        ret = read(pipefd[0], buf, sizeof(buf));
        if (ret == 0)
```

```
        break;
        printf("n=%02d pid=%d read %d bytes from pipe buf[4095]=%c\n", n++, getpid(), ret, buf[4095]);
        write(fd, buf, ret);
    }
    return 0;
}
```

上述代码，创建了三个子进程，第一个子进程每次向管道写入4096字节的A字符，循环10次；第二个子进程向管道写入8192字节（4096*2）的B字符，循环10次；第三个子进程每次向管道写入12288字节（4096*3）的C字符，循环10次。父进程负责从管道里面读取内容，写入到test.txt文件。

由于三个子进程和一个父进程是同时运行的，考虑到进程调度的因素，每次执行写入管道和从管道读取的时序并不完全一样。因此每次执行，产生的test.txt文件也不相同。对于每次写入8KB和每次写入12KB的情况，尽管管道不保证原子性，但是其内容也不是每次都必然会混入其他进程的写入。

多次执行该程序，总会遇到某次8KB或12KB的写入，中间混杂了其他字符。下面的输出是某次执行的结果：

```
0000000 4343 4343 4343 4343 4343 4343 4343 4343
*
0003000 4242 4242 4242 4242 4242 4242 4242 4242
*
0005000 4343 4343 4343 4343 4343 4343 4343 4343
*
0008000 4141 4141 4141 4141 4141 4141 4141 4141
*0009000 4242 4242 4242 4242 4242 4242 4242 4242
*

0010000 4141 4141 4141 4141 4141 4141 4141 4141

*
0015000 4343 4343 4343 4343 4343 4343 4343 4343
*002d000 4242 4242 4242 4242 4242 4242 4242 4242
*

002e000 4141 4141 4141 4141 4141 4141 4141 4141

*
0030000 4242 4242 4242 4242 4242 4242 4242 4242
*
0032000 4141 4141 4141 4141 4141 4141 4141 4141
*
0033000 4242 4242 4242 4242 4242 4242 4242 4242
*
0035000 4141 4141 4141 4141 4141 4141 4141 4141
*
0036000 4242 4242 4242 4242 4242 4242 4242 4242
*
003c000
```

从地址002d000到地址002e000，只有4KB的大小，可是里面的内容却是0x42即B字符。从程序可以得知，B字符每次写入8KB，这里却只有4KB的内容，地址002d000之前是C字符，002e000之后是A字符。唯一的解释就是某次8KB的写入内容被中途打断，混杂了其他进程的写入。

多次执行程序，解读输出的内容，从某些输出中可以看出，8KB的写入和12KB的写入，都不是原子的。

当写入内容长度不超过PIPE_BUF时，内核确保写入操作是原子的这条性质非常重要，尤其是在有多个进程向管道写入的情况下。在不采取其他同步手段的情况下，消息体小于PIPE_BUF时，写入管道是安全的，即使多个进程一起写入也没关系，内核会保证写入内容不会和其他进程的写入内容混在一起。但是如果消息体太大，长度超过了PIPE_BUF，就要警惕，需要采取必要的同步措施，来确保消息内容不会混杂其他进程的消息，否则会导致无法正确解析消息的内容。

9.4 使用管道通信的示例

前文介绍了无名管道pipe和命名管道FIFO，了解了它们的很多性质，但是到目前为止，还没有介绍如何利用管道来实现进程间通信。

下面以FIFO为例，介绍如何使用管道来实现一个客户端/服务器的应用程序，具体流程如图9-12所示。

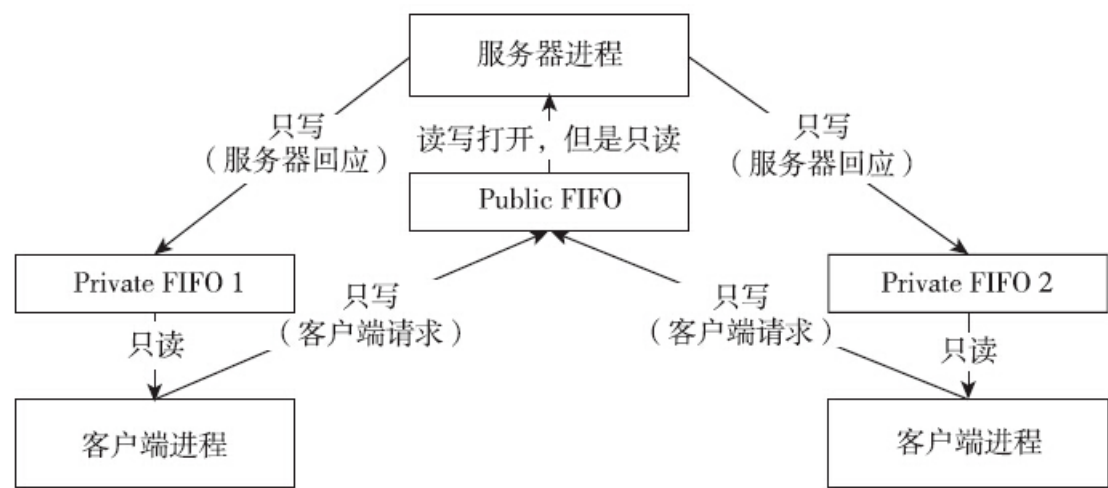


图9-12 使用FIFO实现客户端服务器通信

首先服务器进程会创建一个公开的众所周知的命名管道文件，我们称之为Public FIFO，服务器进程以O_RDWR的模式打开，但是，服务器进程只会从Public FIFO中读取内容，而不会向该命名管道中写入内容。之所以服务器进程要以O_RDWR模式打开（而不是O_RDONLY模式打开），是因为服务器进程是daemon进程，当所有的客户端都关闭曾经打开的Public FIFO，只有自身也以写模式打开Public FIFO，服务器进程的read才不会返回0，而是继续阻塞在管道，等待新的客户发来请求。

```
server_fifo_fd = open(PUBLIC_FIFO_NAME,O_RDWR);
```

这个Public FIFO是众所周知的，所有向服务器进程发送请求的客户端程序都应该知道该Public FIFO的路径。一般来讲，客户端会将自己的请求作为消息体写入Public管道之中。除此以外，客户端会负责创建一个私有的FIFO，用来和服务器进程进行通信。服务器进程从管道中读取了请求之后，就会十分默契地将回应写入到该客户端创建的私有的FIFO中。

问题来了，服务器进程如何知道该往哪个私有的FIFO里面写入，对应的客户端进程才能读到回应信息？方法有很多，比如客户端可以将自己私有的FIFO路径作为请求的一部分，写入到Public FIFO中，服务器进程可以从请求中获得对应客户端的私有FIFO路径。还有一种方法是，私有FIFO有一定的命名规范，比如/tmp/fifo.client_pid，其中client_pid代表客户端进程的进程ID。只要客户端发到Public

FIFO的内容中包含自己的PID，服务器进程就能根据事先的约定找到对应的私有FIFO文件，从而可以将响应写入对应的私有FIFO中。

上述的模型解决了如何利用FIFO编写客户端/服务器程序这个问题。一般来说，不会采用迭代服务的方式。因为某些客户请求处理起来可能非常耗时，那么其他客户端发过来的请求就会被阻塞，得不到及时响应。比较常见的是提供并发服务，即每取出一个请求，就创建一个进程或一个线程来响应该请求。当然还可以提供线程池，让空闲的线程来负责处理客户的请求。

然而，还有一个关键的因素没有讨论。事实上，客户端写入的内容并不是结构化的消息，写入管道之后，客户端进程写入的不过就是字节流。那么，多个进程都向管道写入时，如何正确地区分内容的边界，正确地拣出每个进程的发送内容就成了通信的关键。

一般来说，为了区分内容的边界，有以下办法：

- 写入内容为固定长度。
- 特殊分隔字符。
- 具有长度字段的头。

固定长度的方法最简单，也最容易想到，但是对管道空间的使用效率不高，如图9-13所示。写入的内容长度固定，意味着不得不采用最长消息的长度作为固定长度。对于消息体长度参差不齐，短消息占大多数而最长消息的长度又很长的情况，效率太低，大大降低了管道容纳消息的能力。

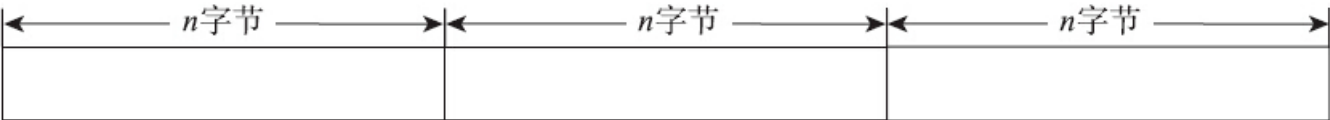


图9-13 固定长度的消息

特殊分隔字符也是一种常用的方法，如图9-14所示。比如事先约定消息的最后一个字符总是换行符。这种方法有几个弊端：第一需要扫描数据，逐个分析字节，直到遇到特殊分隔字符；第二是特殊字符撞车，如果消息体中真的存在事先选定的特殊字符，那就不得不转义。

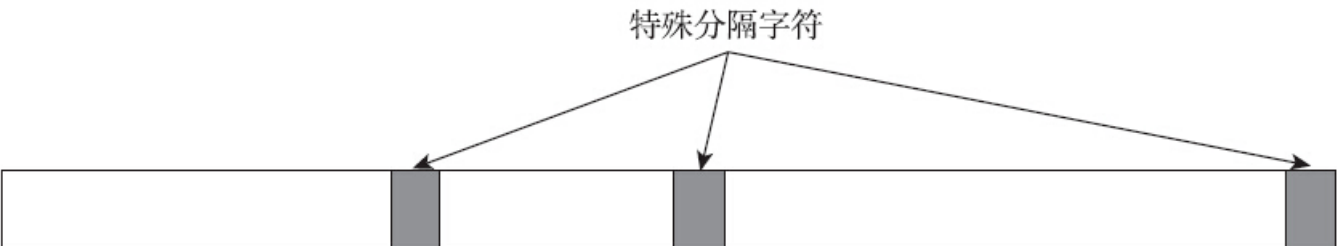


图9-14 特殊分隔字符为结尾的消息

具有长度字段的头是比较推荐的方法，如图9-15所示。管道中提取消息分成两步：

- 1) 提取消息的长度，由于长度字段本身的长度是固定的，所以不会有问题。
- 2) 根据第一步读取的消息长度`len`，读取接下来的`len`字节内容作为消息体。

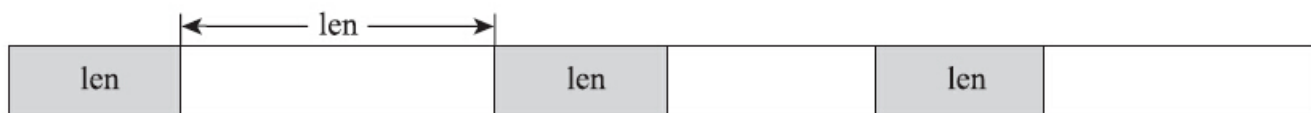


图9-15 以长度字段作为头的消息

值得一提的是，从内核版本3.4开始，内核开始提供**Packet**模式的管道。所谓**Packet**模式，就是写入管道的内容就像是一个**packet**，或者说是一个消息，而不是原始的字节流。

```
ret = pipe2(pipefd,O_DIRECT)
```

当打开管道时，带上**O_DIRECT**标志位，创建的管道就是**Packet**模式的管道，代码如下所示。当然了，老版本的Linux不支持**O_DIRECT**标志位，会返回**EINVAL**错误。

当写入**Packet**模式的管道时，如果写入内容少于**PIPE_BUF**，该内容仍然完全占有一个页面。后面的写入（不管是本进程还是其他进程）不会与上一次的写入共用一个页面。当写入内容大于**PIPE_BUF**时，会分成多个包。

从**Packet**模式管道中读取时，存放读取内容的buffer有**PIPE_BUF**大小肯定足够了。如果指定的buffer太小，小于下一个要取出的**Packet**的大小，管道仍然是取出**Packet**大小，超出buffer的部分会被丢弃掉而不是仍旧留在管道的内存缓冲区。

这种模式从使用内存的角度来看有点浪费，因为不管消息多大，都会至少占有1个页面的大小。但是从编程的角度来看接口更容易使用。

第10章 进程间通信：System V IPC

下面三种类型的进程间通信方法统称为System V IPC：

- System V消息队列

- System V信号量

- System V共享内存

这三种IPC机制的差别很大，之所以将它们放在一起讨论，一个重要的原因是这三种机制是一同被开发出来的。它们最早出现在20世纪70年代末，1983年三者出现在主流的System V Unix系统上，因此这三种机制被统称为System V IPC。

10.1 System V IPC概述

System V IPC相关的接口如表10-1所示。

表10-1 System V IPC编程接口

	消 息 队 列	信 号 量	共 享 内 存
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
关联数据结构	msqid_ds	semid_ds	shmid_ds
创建或打开对象	msgget()	semget()	shmget() + shmat()
关闭对象	无	无	无
控制操作	msgctl()	semctl()	shmctl()
执行 IPC	msgsnd() msgrcv()	semop()	访问共享内存区的内存数据

从作用上看，三种通信机制各不相同，但是从设计和实现的角度来看，还是有很多风格一致的地方。

System V IPC未遵循“一切都是文件”的Unix哲学，而是采用标识符ID和键值来标识一个System V IPC对象。每种System V IPC都有一个相关的get调用（表10-1中的“创建或打开对象”一行），该函数返回一个整型标识符ID，System V IPC后续的函数操作都要作用在该标识符ID上。

System V IPC对象的作用范围是整个操作系统，内核没有维护引用计数。调用各种get函数返回的ID是操作系统范围内的标识符，对于任何进程，无论是否存在亲缘关系，只要有相应的权限，都可以通过操作System V IPC对象来达到通信的目的。

System V IPC对象具有内核持久性。哪怕创建System V IPC对象的进程已经退出，哪怕有一段时间没有任何进程打开该IPC对象，只要不执行删除操作或系统重启，后面启动的进程依然可以使用之前创建的System V IPC对象来通信。

此外，我们也无法像操作文件一样来操作System V IPC对象。System V IPC对象在文件系统中没有实体文件与之关联。我们不能用文件相关的操作函数来访问它或修改它的属性。所以不得不提供专门的系统调用（如msgctl、semop等）来操作这些对象。在shell中无法用ls查看存在的IPC对象，无法用rm将其删除，也无法用chmod来修改它们的访问权限。幸好Linux提供了ipcs、ipcrm和ipcmk等命令来操作这些对象。

由于System V IPC对象不是文件描述符，所以无法使用基于文件描述符的多路转接I/O技术（select、poll和epoll等）。这个缺点会给编程带来一些不便之处。

10.1.1 标识符与IPC Key

System V IPC对象是靠标识符ID来识别和操作的。该标识符要具有系统唯一性。这和文件描述符不同，文件描述符是进程内有效的。一个进程的文件描述符4和另一个进程的文件描述符4可能毫不相干。但是IPC的标识符ID是操作系统的全局变量，只要知道该值（哪怕是猜测获得的）且有相应的权限，任何进程都可以通过标识符进行进程间通信。

三种IPC对象操作的起点都是调用相应的get函数来获取标识符ID，如消息队列的get函数为：

```
int msgget(key_t key, int msgflg);
```

其中第一个参数是key_t类型，它其实是一个整型的变量。IPC的get函数将key转换成相应的IPC标识符。根据IPC get函数中的第二个参数oflag的不同，会有不同的控制逻辑，如表10-2所示。

表10-2 创建或打开一个IPC对象的逻辑

oflag 参数	key 不存在	key 已经存在
无特殊标志	出错返回 -1(ENOENT)	成功返回 0，获取到已有标识符
IPC_CREAT	成功返回 0，创建新标识符	成功返回 0，获取到已有标识符
IPC_CREAT IPC_EXCL	成功返回 0，创建新标识符	出错返回 -1(EEXIST)

因为key可以产生IPC标识符，所以很容易产生一种误解，就是同一个key调用IPC的get函数总是返回同一个整型值。实际上并非如此。在IPC对象的生命周期中，key到标识符ID的映射是稳定不变的，即同一个key调用get函数，总是返回相同的标识符ID。但是一旦key对应的IPC对象被删除或系统重启后，则重新使用key创建的新的IPC对象被分配的标识符很可能是不同的。

不同进程可通过同一个key获取标识符ID，进而操作同一个System V IPC对象。那么现在问题就演变成了如何选择key。

对于key的选择，存在以下三种方法。

第一种方法是随机选择一个整数值作为key值（如图10-1所示）。作为key值的整数通常被放在一个头文件中，所有使用该IPC对象的程序都要包含该头文件。需要注意的是，要防止无意中选择了重复的key值，从而导致不需要通信的进程之间意外通信，以致引发程序混乱。一个技巧是将项目要用到的所有key放入同一个头文件中，这样就可以方便地检查是否有重复的key值。

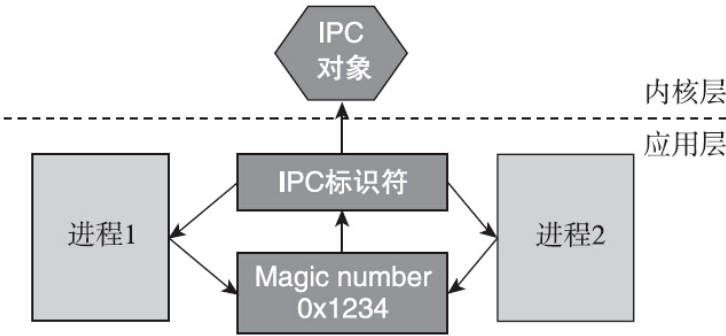


图10-1 使用magic number作为key

第二种方法是使用IPC_PRIVATE，使用方法如下：

```
id = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);
```

这种方法无须指定IPC_CREATE和IPC_EXCL标志位，就能创建一个新的IPC对象。使用IPC_PRIVATE时总是会创建新的IPC对象，从这个角度看将其称之为IPC_NEW或许更合理。

不过，使用IPC_PRIVATE来得到IPC标识符会存在一个问题，即不相干的进程无法通过key值得到同一个IPC标识符。因为IPC_PRIVATE总是会创建一个新的IPC对象，如图10-2所示。因此IPC_PRIVATE一般用于父子进程，父进程调用fork之前创建IPC对象，创建子进程后，子进程也就继

承了IPC标识符，从而父子进程可以通信。当然无亲缘关系的进程也可以使用IPC_PRIVATE，只是稍微麻烦了一点，IPC对象的创建者必须想办法将IPC标识符共享出去，让其他进程有办法获取到，从而通过IPC标识符进行通信。

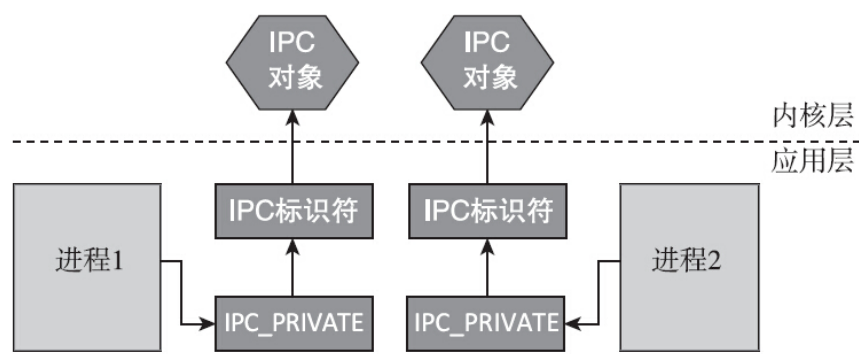


图10-2 IPC_PRIVATE总是创建新的IPC对象

第三种方法是使用ftok函数，根据文件名生成一个key。ftok是file to key的意思，多个进程通过同一个路径名获得相同的key值，进而得到同一个IPC标识符。其使用方法如图10-3所示。

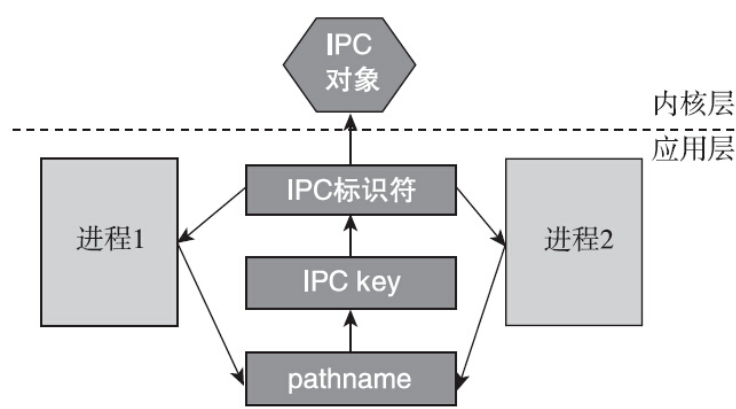


图10-3 根据文件获得key，进而获得IPC标识符

ftok函数接口的定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

在Linux实现中，该接口把通过path-name获取的信息和传入的第二个参数的低8位糅合在一起，得到一个整型的IPC key值，如图10-4所示。需要注意的是，pathname对应的文件必须是存在的。

这个函数在Linux上的实现是：按照给定的路径名，获取到文件的stat信息，从stat信息中取出st_dev和st_ino，然后结合给出的proj_id，按照图10-4所示的算法获取到32位的key值。

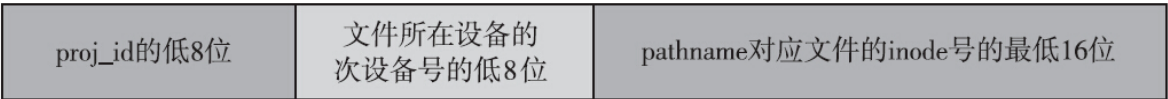


图10-4 ftok生成键值的算法

可以用程序来验证，代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
int main(int argc, const char* argv[])
{
    struct stat stat_buf;
    if(argc != 2)
    {
        fprintf(stderr, "Usage : ftok <pathname>");
        return 1;
    }
    stat(argv[1], &stat_buf);
```

```
key_t key = ftok(argv[1],0x1234);
printf("st_dev : %lx, st_inode : %lx , key : %x \n",
      stat_buf.st_dev,stat_buf.st_ino,key);
return 0;
}
```

执行情况如下：

```
./ftok_test.c
st_dev : 801, st_inode : 240cb4 , key : 34010cb4
```

观察上面的加粗部分，可以看出key确实是按照图10-4所示的数据来源糅合而得到的。即使是ftok函数的第二个参数相同，也很难出现两个文件映射出同一个key值的情况。这里说的是很难，而不是绝对不会，因为这种情况是有可能发生的。这种冲突的出现需要同时满足下面三个条件：

- 两个文件所属文件系统所在磁盘的次设备号的低8位相同。
- 两个文件在各自的文件系统上的inode的最低16位也相同。
- 两个进程分别选择同一个proj_id来调用ftok（）来获取key值。

虽然理论上是存在key值冲突的可能，但是实际上，不同的文件通过ftok函数产生出冲突的key值的可能性太低，除非刻意构造这种冲突，否则很难出现。因此使用ftok函数来获取key值是编程中常用的方法。

10.1.2 IPC的公共数据结构

三种System V IPC对象有很多共性，从代码层面上看也有很多公共的部分。权限结构就是其中一个。IPC的权限结构至少包括如下成员：

```
struct ipc_perm{
    key_t key;
    uid_t uid;
    gid_t gid;
    uid_t cuid;
    gid_t cgid;
    mode_t mode;
    ulong_t seq ;
};
/*消息队列控制相关的结构体

/
struct msqid_ds {
    struct ipc_perm msg_perm;
    ...
}
/*信号量控制相关的结构体

*/
struct semid_ds {
    struct ipc_perm sem_perm;
    ...
}
/*共享内存控制相关的结构体

*/
struct shmid_ds {
    struct ipc_perm shm_perm;
    ...
}
```

uid和gid字段用于指定IPC对象的所有权。cuid和cgid字段保存着创建该IPC对象的进程的有效用户ID和有效组ID。初始情况下，用户ID（uid）和创建者ID（cuid）的值是相同的。它们都是调用进程的有效ID。但是创建者ID（cuid）是不可以改变的，而所有者ID则可以通过IPC_SET来改写。下面的代码演示了如何修改共享内存的uid字段：

```
struct shmid_ds shm_ds;
if(shmctl(id,IPC_STAT,&shm_ds) == -1)
{
    /*error handler*/
}
shm_ds.shm_perm.uid = newuid;
if(shmctl(id,IPC_SET,&shm_ds) == -1)
{
    /*error handle*/
}
```

mode是用来控制读写权限的。所有的System V IPC对象都不具备执行权限，只有读写权限。其中对于信号量而言，写权限意味着修改权限。IPC对象的权限控制见表10-3。

表10-3 IPC对象的权限控制

权 限	标 志 位	位
用户读	S_IRUSR	0400
用户写	S_IWUSR	0200
用户组读	S_IRGRP	0040
用户组写	S_IWGRP	0020
其他读	S_IROTH	0004
其他写	S_IWOTH	0002

和文件的权限有点类似，IPC对象的权限被分成了三类：owner、group和other。创建对象时可以为各个类别设定不同的访问权限，代码如下所示：

```
msg_id = msgget(key,IPC_CREAT | S_IRUSR | S_IWUSR |S_IRGRP);
msg_id = msgget(key,IPC_CREAT | 0640);
```

当一个进程尝试对IPC对象执行某种操作的时候，首先会检查权限。检查的逻辑如下：

- 如果进程是特权进程，那么进程拥有对IPC对象的所有权。
- 如果进程的有效用户ID与IPC对象的所有者或创建者ID匹配，那么会将对象的owner的权限赋给进程。
- 如果进程的有效用户ID或任意一个辅助组ID与IPC对象的所有者组ID或创建者组ID匹配，那么会将IPC对象的group的权限赋予进程。
- 否则，将IPC对象的other权限赋予进程。

数据结构ipc_perm中的key和seq也很有意思。key比较简单，就是调用get函数创建IPC对象时传递进去的key值。如果key的值是IPC_PRIVATE，则实际的key值是0。

和key相比，成员变量seq就不那么好理解了。进程分配文件描述符时采用的是最小可用算法。比如文件描述符5曾经被分配给文件A，但是很快进程关闭了文件A。如果进程尝试打开另外一个文件，此时如果5是最小可用的槽位，那么新打开文件的文件描述符就是5。但是IPC对象的标识符ID分配不能采用这个算法。因为多个进程要通过标识符ID来通信，而标识符ID是整个系统内有效的。如果采用最小可用的算法，一般来讲，IPC对象的个数不会太多，那么这个数字很容易就被猜到了。举例来说，如果存在一个恶意程序要攻击消息队列，它只需尝试很小范围内的数字，就可以猜到IPC对象的标识符ID，进而偷偷取走消息队列里面的信息。

内核针为每一种System V IPC维护了一个ipc_ids类型的结构体。该结构体的组成如图10-5所示。

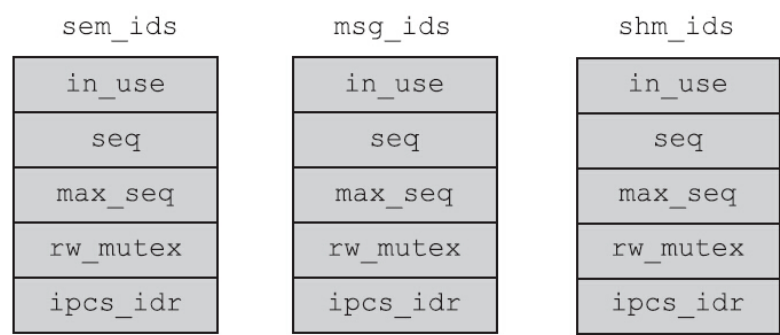


图10-5 System V IPC的ipc_ids数据结构

上述结构体中in_use字段记录的是系统当前在用的IPC个数。因此创建IPC对象时，该值会加1；销毁IPC对象时，该值会减去1。

结构体中seq字段记录了开机以来创建该IPC对象的流水号。创建时seq的值自加，但是销毁的时候seq的值并不会自减。seq的值随着该种IPC对象的创建而单调地递增，直到递增到上限（max_seq），再溢出回绕，重新从0开始。

当需要创建新的IPC对象时，三种IPC对象的创建都会走到ipc_addid函数处，如图10-6所示。

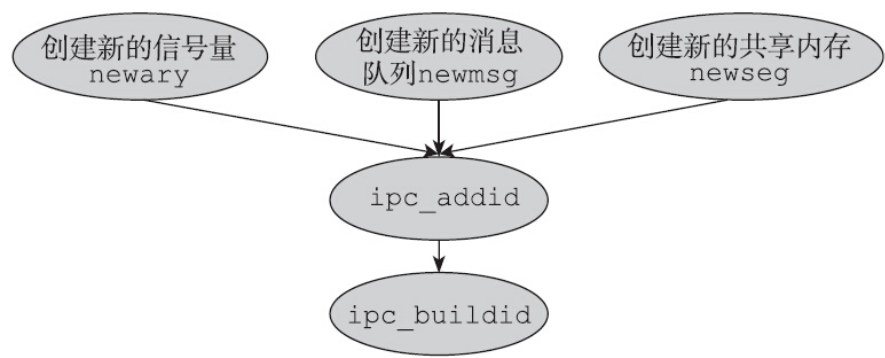


图10-6 为新的IPC对象生成标识符ID

ipc_addid函数会初始化IPC对象的很多成员变量，比如权限相关的uid、gid、cuid和cgid，也会维护该IPC对象的seq值。

```

int ipc_addid(struct ipc_ids* ids, struct kern_ipc_perm* new, int size)
{
    uid_t euid;
    gid_t egid;
    int id, err;
    /*用户设置的
  
```

IPC对象的上限，不能超过系统硬上限

IPCMNI，即

```
32768*/
if (size > IPCMNI)
    size = IPCMNI; /*如果系统中已经存在的
```

IPC对象超过了个数上限，则返回失败

```
*/
if (ids->in_use >= size)
    return -ENOSPC;
spin_lock_init(&new->lock);
new->deleted = 0;
rcu_read_lock();
spin_lock(&new->lock);
/*通过
```

idr管理，调用

idr_get_new获得一个空闲的槽位

```
*/
err = idr_get_new(&ids->ipcs_idr, new, &id);
if (err) {
    spin_unlock(&new->lock);
    rcu_read_unlock();
    return err;
}
/*系统当前在用的
```

IPC对象加

```
1*/
ids->in_use++;
/*设置创建者
```

ID和

```
owner ID*/
current_euid_egid(&euid, &egid);
new->cuuid = new->uid = euid;
new->gid = new->cgid = egid;
/*seq的值自加，如果大于
```

seq_max，则溢出回绕至

```
0*/
new->seq = ids->seq++;
if (ids->seq > ids->seq_max)
    ids->seq = 0;
new->id = ipc_buildid(id, new->seq);
return id;
}
```

前面提到，内核分配IPC对象标识符的时候，使用的并不是最小可用算法，其使用的算法如下：

```
#define IPCMNI 32768
#define SEQ_MULTIPLIER (IPCMNI)
static inline int ipc_buildid(int id, int seq)
{
    return SEQ_MULTIPLIER * seq + id;
}
```

上面公式中的id就是最小可用的槽位，而seq是开机以来内核创建IPC对象的流水号。因此，返回的ID是一个比较大的值。仍然以消息队列为例，如果开机后，消息队列为空，创建的第一个消息队列的标识符必然为0，而创建的第二个消息队列和第三个消息队列的值则为：

```
32768 * 1 + 1 = 32769
32768 * 2 + 2 = 65538
```

根据上面的讨论可知，IPC对象的标识符ID虽然是通过get函数来获得的，但是和key值并不存在永久的对应关系，即不存在公式可以通过key值来计算出标识符ID。内核仅仅是关联了两者。重启系统之后，或者删除IPC对象之后，根据相同的key值再次创建，得到的标识符ID很可能并

不相同。

内核面临着如何根据IPC对象的标识符ID，快速地找到内核中的IPC对象的难题，根据前面的计算公式，不难做到：

```
slot_index = 标识符

ID % SEQ_MULTIPLIER
```

这个公式透漏出了一个**问题**：整个系统内，每一种IPC对象的槽位有限，最多有IPCMIN个槽位。在ipc_addid函数中也证实了这一点，系统的硬上限为IPCMNI，即32768。这个限制就决定了不能无限制地创建IPC对象。

10.2 System V消息队列

第9章介绍的管道和FIFO都是字节流的模型，这种模型不存在记录边界。如果从管道里面读出100个字节，你无法确认这100个字节是单次写入的100字节，还是分10次每次10字节写入的，你也无法知晓这100个字节是几个消息。管道或FIFO里的数据如何解读，完全取决于写入进程和读取进程之间的约定。

从这个角度上讲，System V消息队列和POSIX消息队列都是优于管道和FIFO的。原因是消息队列机制中，双方是通过消息来通信的，无需花费精力从字节流中解析出完整的消息。

System V消息队列比管道或FIFO优越的第二个地方在于每条消息都有type字段，消息的读取进程可以通过type字段来选择自己感兴趣的消息，也可以根据type字段来实现按消息的优先级进行读取，而不一定要按照消息生成的顺序来依次读取。

内核为每一个System V消息队列分配了一个msg_queue类型的结构体，其成员变量和各自的含义如下所示：

```
struct msg_queue {
    struct kern_ipc_perm q_perm;
    time_t q_stime; /* 上一次

msgsnd的时间

*/
    time_t q_rtime; /* 上一次

msgrcv的时间

*/
    time_t q_ctime; /* 属性变化时间

*/
    unsigned long q_cbytes; /* 队列当前字节总数

*/
    unsigned long q_qnum; /* 队列当前消息总数

*/
    unsigned long q_qbytes; /* 一个消息队列允许的最大字节数

*/
    pid_t q_lspid; /* 上一个调用
```

msgsnd的进程

```
ID*/
    pid_t q_lrpid;           /*上一个调用
```

msgrcv的进程

```
ID*/
    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};
```

大部分字段的含义都是比较好理解的，后面遇到相关内容的时候会详细讲述这些字段。

10.2.1 创建或打开一个消息队列

消息队列的创建或打开是由msgget函数来完成的，成功后，获得消息队列的标识符ID，函数接口定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

msgget函数中两个参数的含义前面已经讲述过了，在此就不再赘述。当调用成功时，返回消息队列的标识符，后续的msgsnd、msgrcv和msgctl函数都通过该标识符来操作消息队列。当函数调用失败时，返回-1，并且设置相应的errno。常见的errno如表10-4所示。

表10-4 msgget出错情况说明

errno	说 明
ENOENT	对应 key 值的消息队列不存在，并且没有设定 IPC_CREAT 标志位
EACCES	存在 key 值对应的消息队列，但是没有权限打开消息队列
EEXIST	存在 key 值对应的消息队列，但是同时设置了 IPC_CREAT 和 IPC_EXCL 标志位
ENOSPC	需要创建消息队列，但是超过了系统允许创建消息队列的上限。上限值为 MSGMNI
ENOMEM	需要创建消息队列，但是系统已经没有足够的内存

关于创建消息队列，一个很容易想到的问题是：操作系统到底允许创建多少个消息队列？

表10-4中提到，当errno等于ENOSPC时，表示创建的消息队列超过了上限值MSGMNI。有三种方法可以查看系统消息队列个数的上限，如下所示。

方法一：通过procfs查看。

```
cat /proc/sys/kernel/msgmni
3969
```

方法二：通过sysctl查看。

```
sysctl kernel.msgmni
kernel.msgmni = 3969
```

方法三：通过ipcs命令查看。

```
ipcs -q -l
----- Messages Limits -----
max queues system wide = 3969
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

操作系统会根据系统的硬件情况（主要是内存大小），计算出一个合理的上限值，因此不同的硬件环境下，该值是不同的。当然无论该值设置为多少，内核都存在硬上限IPCMNI（32768）。

可以通过如下的手段，修改msgmni的值，从而允许创建更多的消息队列。

方法一：通过procfs来修改。

```
echo 20000 > /proc/sys/kernel/msgmni
cat /proc/sys/kernel/msgmni
20000
```

方法二：通过sysctl-w来修改。

```
sysctl -w kernel.msgmni=20000
```

上述两种方法都是立即生效，但是一旦系统重启，设置就失去了。要想确保重启后依然有效，需要将配置写入/etc/sysctl.conf。

```
kernel.msgmni=20000
```

注意写入/etc/sysctl.conf并不会立即生效，需要执行sysctl-p重新加载，改变方能生效。

10.2.2 发送消息

获取到消息队列的标识符之后，可以通过调用msgsnd函数向队列中插入消息。内核会负责将消息维护在消息队列中，等待另外的进程来取走消息，从而完成通信的全过程。

msgsnd函数的定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

其中msqid是由msgget返回的标识符ID。

参数msgp指向用户定义的缓冲区。它的第一个成员必须是一个指定消息类型的long型，后面跟着消息文本的内容。通常其定义如下：

```
struct msgbuf {
    long mtype;           /*消息类型，必须大于

0*/
    char mtext[1];       /*消息体，不一定是字符数组，可以是任意结构

*/
};
```

每条消息只能存放一个字符？并非如此。事实上可以是任意结构，mtext是由程序员定义的结构，其长度和内容都是由程序员控制的，只要发送方和接收方约定好即可。比如可以将结构体定义如下：

```
struct private_buf {
    long mtype;
    struct pirate_info {
        /*定义你需要的成员变量

*/
    } info;
};
```

第三个参数msgsz指定了mtext字段中包含的字节数。消息队列单条消息的大小是有上限的，上限值为MSGMAX，记录在/proc/sys/kernel/msgmax中：

```
cat /proc/sys/kernel/msgmax
8192
sysctl kernel.msgmax
kernel.msgmax = 8192
```

如果消息的长度超过了MSGMAX，那么msgsnd函数返回-1，并置errno为EINVAL。

下面以发送字符串消息为例，介绍msgsnd函数所需的步骤：

- 1) 因为glibc并未定义msgbuf结构体，因此首先要定义msgbuf结构体。
- 2) 分配一个类型为msgbuf，长度足以容纳字符串的缓冲区mbuf。
- 3) 将message的内容拷贝到mbuf->mtext中去。
- 4) 在mbuf->mtype中设置消息类型。
- 5) 调用msgsnd发送消息。
- 6) 释放mbuf。

注意两点，即要对msgsnd进行错误检测和及时释放mbuf，以防止内存泄漏。

最后一个参数msgflg是一组标志位的位掩码，用于控制msgsnd的行为。目前只定义了IPC_NOWAIT一个标志位。

IPC_NOWAIT表示执行一个无阻塞的发送操作。当没有设置IPC_NOWAIT标志位时，如果消息队列满了，那么msgsnd函数就会陷入阻塞，直到队列有足够的空间来存放这条消息为止。但是如果设置了IPC_NOWAIT标志位，那么msgsnd函数就不会陷入阻塞了，而是立刻返回失败，并置errno为EAGAIN。

等一下，这里好像提到了消息队列满。什么情况下，消息队列才能被称为是满的？

任何一个消息队列，容纳的字节数是有上限的。这个上限值为MSGMNB，该值被记录在/proc/sys/kernel/msgmnb中：

```
cat /proc/sys/kernel/msgmnb
16384
sysctl kernel.msgmnb
kernel.msgmnb = 16384
```

内核中消息队列对应的数据结构msg_queue中维护有当前字节数、当前消息数及允许的最大字节数等信息：

```
struct msg_queue {

    ...
    time_t q_stime;           /*最后调用

msgsnd的时间

*/
    unsigned long q_cbytes;    /*消息队列当前字节的总数

*/
    unsigned long q_qnum;      /*消息队列当前消息的个数

*/
    unsigned long q_qbytes;    /*消息队列允许的消息最大字节数

*/
    pid_t q_lspid;            /*最后调用

msgsnd的进程

ID*/
    ...
}
```

检查消息队列是否满的逻辑非常简单，内核判断能否立刻发送消息的逻辑如下：

```
if (msgsz + msq->q_cbytes <= msq->q_qbytes &&
    1 + msq->q_qnum <= msq->q_qbytes) {
    break;
}
```

如果同时满足以下两个条件，则可以立即发送消息，无须阻塞：

- 当前消息的字节数（msgsz）加上消息队列当前字节的总数（msq->q_cbytes）不大于消息队列允许的最大字节数（msq->q_qbytes）。
- 消息队列当前消息的个数加上1不大于消息队列容许的最大字节数（msq->q_qbytes）。

第二个条件看起来很奇怪的，其实这个条件是用来防范空消息的：发送的消息只有mtype字段，消息体正文mtext都是空的。

不满足上述两个条件的话，msgsnd函数会根据是否设置了IPC_NOWAIT标志位来决定是陷入阻塞还是立刻返回失败。

如果因消息队列满而陷入阻塞，msgsnd系统调用则可能会被信号中断，当这种情况发生时，msgsnd总是返回EINTR错误。注意，无论在建立信号处理函数的时候，是否设置了SA_RESTART标志位，msgsnd系统调用都不会自动重启。

无论是否经过阻塞，只要没有出错返回，调用msgsnd都需要执行下面的操作：

```
/*将最后调用

msgsnd的进程

ID更新到消息队列的

q_lspid成员变量中

*/msg->q_lspid = task_tgid_vnr(current);/*将最后调用

msgsnd的时间更新到消息队列的

q_stime成员变量中

*/
msg->q_stime = get_seconds();/*如果有进程正在等待该消息，则就地消化，无须进入消息队列

*/if (!pipelined_send(msg, msg)) {
    /*将消息链入消息队列的链表中

    */
    list_add_tail(&msg->m_list, &msg->q_messages);
    /*更新消息队列当前消息的字节数

    */
    msg->q_cbytes += msgsz;
    /*更新消息队列当前消息的总数

    */
    msg->q_qnum++;
    /*更新命名空间内，所有消息队列的总字节数和消息总个数

    */
    atomic_add(msgsz, &ns->msg_bytes);
    atomic_inc(&ns->msg_hdrs);
}
```

pipelined_send函数用于检测是否有进程正在等待该消息，如果有的话，消息无须进入消息队列，而是“就地消化”，皆大欢喜。如果没有等待该消息的进程，则消息就不得不进入消息队列，等待“有缘人”来提取。

至此，msgsnd函数的使用和流程基本介绍完毕，如果执行成功，则msgsnd返回0，如果失败，msgsnd则返回-1，并置errno。

下面分析一下函数的返回值和常见错误。msgsnd函数不同于文件的write函数，write函数操作的是字节流，存在部分成功的概念，所以成功时，返回的是写入的字节个数；但是msgsnd函数操作的是封装好的消息，不成功则成仁，不存在部分成功的情况。所以其成功时，msgsnd函数返回0，失败时，msgsnd函数返回-1，并且设置errno。常见的出错情况如表10-5所示。

表10-5 msgsnd出错情况说明

errno	说 明
EACCESS	无相应权限
EAGAIN	消息队列已满，并且设置了IPC_WAIT标志位
EIDRM	消息队列已经从系统中删除，不复存在
EINTR	msgsnd 被信号中断
EINVAL	标识符无效，mtype 小于 1，msgsz 小于 0 或大于上限 MSGMAX

几乎所有的出错情况前面都已经介绍过了，除了EIDRM。这是消息队列和信号量的共同缺陷。当一个进程操作消息队列时，另外一个进程可能已经删除该消息队列了。对于IPC对象（共享内存除外），内核并没有维护引用计数，删除行为是说删就删，于是msgsnd调用就会收到

EIDRM的错误。

删除消息队列是一个编程难点，难就难在确定删除的时机。多个进程需要从逻辑上确定谁是最后一个访问消息队列的进程，然后由它来负责删除消息队列。

10.2.3 接收消息

有发送就要有接收，没有接收者的消息是没有意义的。System V消息队列用msgrcv函数来接收消息。

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
               long msgtyp,int msgflg);
```

其中前三个参数与msgsnd的含义是一致的。msgrcv调用进程也需要定义结构体，而结构体的定义要和发送端的定义一致，并且第一个字段必须是long类型，代码如下所示：

```
struct private_buf {long mtype;struct pirate_info {    /*定义你需要的成员变量

*/} info;
};
```

对于具有固定长度的消息体来讲，只要发送方和接收方的结构体达成一致，就不会存在风险。但是如果消息体是变长的，情况就复杂了点。因为不能预先得知收到消息体的长度，因此接收端的缓冲区要足够大，防止消息队列中的消息长度大于缓冲区的大小。

Msgrcv函数的第4个参数msgtyp是消息队列的精华，提取消息时，可以选择进程感兴趣的消息类型。正是基于这个参数，读取消息的顺序才无须和发送顺序一致，进而可以演化出很多用法。msgtype与提取消息的行为关系如表10-6所示。

表10-6 msgtype与提取消息的行为

msgtype	动 作
0	从消息队列中取出第一条消息
(续)	
msgtype	动 作
>0	有 MSG_EXCEPT 标志位：从消息队列中取出 mtype 等于 msgtyp 的第一条消息
	无 MSG_EXCEPT 标志位：从消息队列中取出 mtype 不等于 msgtyp 的第一条消息
<0	从消息队列中取出 mtype 最小，并且值小于或等于 msgtyp 绝对值的第一条消息

当msgtyp等于0时，行为模式是先入先出的模式。最先进入消息队列的消息被取出。

当msgtyp小于0时，行为模式是优先级消息队列。mtype的值越低，其优先级越高，越早被取出。

当msgtyp的值大于0时，会将消息队列中第一条mtype值等于msgtyp的消息取出。通过指定不同的msgtyp，多个进程可以在同一个消息队列中挑选各自感兴趣的消息。一种常见的场景是各个进程提取和自己进程ID匹配的消息。

第5个参数是可选标志位。msgrcv函数有3个可选标志位。

- IPC_NOWAIT：如果消息队列中不存在满足msgtyp要求的消息，默认情况是阻塞等待，但是一旦设置了IPC_NOWAIT标志位，则立即返回失败，并且设置errno为ENOMSG。
- MSG_EXCEPT：这个标志位是Linux特有的，只有当msgtyp大于0时才有意义，含义是选择mtype!=msgtyp的第一条消息。
- MSG_NOERROR：前面也提到过，在消息体变长的情况下，可能事前并不知道消息体的大小，尽管要求maxmsgsz应尽可能地大，但是仍然存在maxmsgsz小于消息体大小的可能。如果发生这种情况，默认情况是返回错误E2BIG，但是如果设置了MSG_NOERROR标志位，情况就不同了，此时会将消息体截断并返回。

msgrcv函数调用成功时，返回消息体的大小；失败时返回-1，并且设置errno。大部分出错情况和msgsnd函数类似，比较特殊的错误码是E2BIG和ENOMSG，刚才都已经讨论过了，这里不再赘述。另外msgrcv函数和msgsnd函数一样，如果被信号中断，则不会重启系统调用，哪怕安装信号时设置了SA_RESTART标志位。

System V消息队列存在一个问题，即当消息队列中有消息到来时，无法通知到某进程。消息队列的读取者进程，要么以阻塞的方式调用`msgrcv`函数，阻塞在消息队列上直到消息出现；要么以非阻塞（`IPC_NOWAIT`）的方式调用`msgrcv`函数，失败返回，过段时间再重试，除此以外并无好办法。阻塞或轮询，这就意味着一个进程或线程不得不无所事事，盯在该消息队列上，这给编程带来了不便。

如果System V消息队列是文件，能支持`select`、`poll`和`epoll`等I/O多路转接函数，一个进程就能同时监控多个文件（或者多个消息队列），提供更灵活的编程模式。可惜的是，System V消息队列并非文件，不支持I/O多路转接函数。第11章中可以看到POSIX消息队列在这个方面所做的改进。

10.2.4 控制消息队列

msgctl函数可以控制消息队列的属性，其接口定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

该函数提供的功能取决cmd字段，msgctl支持的操作如表10-7所示。

表10-7 msgctl支持的命令

cmd	描 述
IPC_STAT	获取消息队列的属性信息
IPC_SET	设置消息队列的属性
IPC_RMID	删除消息队列

1.IPC_STAT

为了获取消息队列的属性信息或设置属性，必须要有一个用户态的数据结构来描述消息队列的属性信息，这个数据结构就是msqid_ds结构体，其大部分字段和内核的msg_queue结构体相对应。注意，msqid_ds结构体中包含下面的成员变量。在编程中，只要包含了对应的头文件，就可以直接使用该结构体。

```
#include <sys/msg.h>
struct msqid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t msg_stime; /*最后一次调用

msgsnd的时间

*/
    time_t msg_rtime; /*最后一次调用

msgrcv的时间

*/
    time_t msg_ctime; /*属性发生变化的时间

*/
    unsigned long __msg_cbytes; /*消息队列当前的字节总数

*/
    msgqnum_t msg_qnum; /*消息队列当前消息的个数

*/
    msglen_t msg_qbytes; /*消息队列允许的最大字节数

*/
    pid_t msg_lspid; /*最后一次调用

msgsnd的进程

ID */
    pid_t msg_lrpid; /*最后一次调用

msgrcv的进程

ID*/
};
```

几乎全部的字段都和内核的msg_queue相对应，而其对应的字段的含义在前面都已经介绍过了，此处不再赘述。在使用时，我们可以通过下面的简单代码来获取到消息队列的属性：

```
struct msqid_ds buf ;           /*注意包含头文件

*/
msgctl(mid,IPC_STAT,&buf);      /*省略

error handle*/
printf("

current # of messages in queue is %d\n"

,buf.msg_qnum);
```

2.IPC_SET

消息队列开放出了4个可以设置的属性。

- msg_perm.uid
- msg_perm.gid
- msg_perm.mode
- msg_qbytes

设置方法一般首先调用IPC_STAT获取到当前的设置，然后修改4个属性中的某个或某几个属性，最后调用IPC_SET，代码如下所示：

```
struct msqid_ds buf ;           /*注意包含头文件

*/
msgctl(mid,IPC_STAT,&buf);      /*省略

error handle*/
buf.msg_qbytes = NEW_VALUE;
msgctl(mid,IPC_SET,&buf);
```

3.IPC_RMID

IPC_RMID命令用于删除与标识符对应的消息队列。由于IPC对象并无引用计数的机制，因此只要有权限，可以说删就删，而且是立刻就删。消息队列中的所有消息都会被清除，相关的数据结构被释放，所有阻塞的msgsnd函数和msgrcv函数会被唤醒，并返回EIDRM错误。

10.3 System V信号量

10.3.1 信号量概述

System V信号量又被称为System V信号量集，事实上信号量集的叫法更符合实际情况。信号量的作用和消息队列不太一样，消息队列的作用是进程之间传递消息。而信号量的作用是为了同步多个进程的操作。

信号量是由E.W.Dijkstra为互斥和同步的高级管理提出的概念。它支持两种原子操作，wait和signal。wait还可以称为down、P或lock，signal还可以称为up、V、unlock或post。其作用分别是原子地增加和减少信号量的值。

一般来说，信号量是和某种预先定义的资源相关联的。信号量元素的值，表示与之关联的资源的个数。内核会负责维护信号量的值，并确保其值不小于0。

信号量上支持的操作有：

- 将信号量的值设置成一个绝对值。
- 在信号量当前值的基础上加上一个数量。
- 在信号量当前值的基础上减去一个数量。
- 等待信号量的值等于0。

在上述操作中，后两个可能会陷入阻塞。在第三种情况中，当信号量的当前值小于要减去的值时，操作会陷入阻塞。当信号量的值不小于要减去的值时，内核会唤醒阻塞进程。在第四种情况中，如果当前信号量的值不为0，该操作会陷入阻塞，直到信号量的值变为0为止。

这些操作看似没有什么意义，但是一旦将信号量和某种资源关联起来，就起到了同步使用某种资源的功效，请看表10-8。

表10-8 信号量与资源管理

信号量操作	语 义
将信号量的值设为某绝对值	初始化资源的个数为某绝对值
在信号量当前值的基础上加上一个数量 N	释放 N 个资源
在信号量当前值的基础上减去一个数量 M	申请 M 个资源，可能因资源不够而陷入阻塞
等待信号量的值变为 0	等待可用资源个数变为 0，可能会陷入阻塞

使用最广泛的信号量是二值信号量（binary semaphore）。对于这种信号量而言，它只有两种合法值：0和1，对应一个可用的资源。若当前有资源可用，则与之对应的二值信号量的值为1；若资源已被占用，则与之对应的二值信号量的值为0。当进程申请资源时，如果当前信号量的值为0，那么进程会陷入阻塞，直到有其他进程释放资源，将信号量的值加1才能被唤醒。

从这个角度看，二值信号量和互斥量所起的作用非常类似。那信号量和互斥量有何不同之处呢？

互斥量（mutex）是用来保护临界区的，所谓临界区，是指同一时间只能容许一个进程进入。而信号量（semaphore）是用来管理资源的，资源的个数不一定是1，可能同时存在多个一模一样的资源，因此容许多个进程同时使用资源。

有个很有意思的卫生间理论可以用来阐述互斥量和信号量的区别。互斥量好比是一把卫生间的钥匙，卫生间只有一个，钥匙也只是一把。需要使用卫生间时，首先要去钥匙存放处取走钥匙，当使用完卫生间时，要将钥匙归还到钥匙存放处。如果某人需要使用卫生间，发现钥匙存放处没有钥匙，那么他就需要等待，直到卫生间的当前使用者将钥匙归还。

假设后来买了一套豪宅，家里有8个一模一样的卫生间和8把通用的钥匙。这时信号量就横空出世了。信号量的值的含义是当前可用的钥匙数，最初有8把钥匙放在钥匙存放处。当同时使用卫生间的人数小于或等于8时，大家都可以拿到一把钥匙，各自使用各自的卫生间。但是到第9个人和第10个人要使用卫生间时，发现已经没有钥匙了，所以他们就不得不等待了。

从上面的讨论看，信号量是互斥量的一个扩展，由于资源数目增多，增强了并行度。但是这仅仅是一个方面。更重要的区别是，互斥量和信号量解决的问题是不同的。

互斥量的关键在于互斥、排它，同一时间只允许一个线程访问临界区。这种严格的互斥，决定了了解铃还须系铃人，即加锁进程必然也是解锁进程，代码如下所示：

进程

1

2

pthread_mutex_lock();

/*安全地访问临界区

*/

pthread_mutex_unlock();

*/

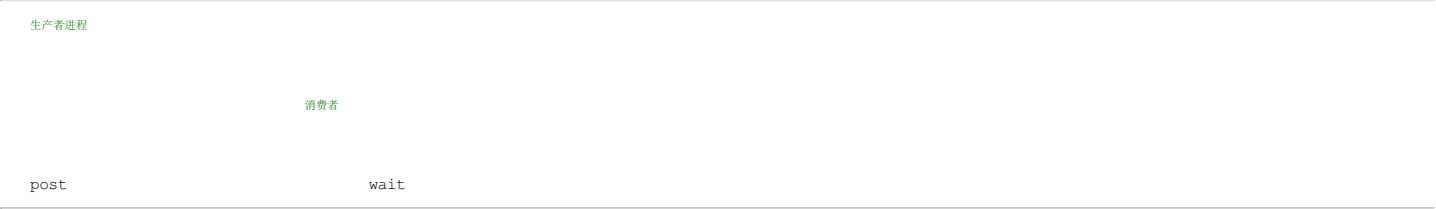
pthread_mutex_lock();

/*安全地访问临界区

*/

Pthread_mutex_unlock();

而信号量的关键在于资源的多少和有无。申请资源的进程不一定要释放资源，信号量同样可以用于生产者-消费者的场景。在这种场景下，生产者进程只负责增加信号量的值，而消费者进程只负责减少信号量的值。彼此之间通过信号量的值来同步。



和二值信号量相比，System V信号量在两个维度上都做了扩展。

第一，资源的数目可以是多个。资源个数超过1个的信号量称为计数信号量（counting semaphore）。

第二，允许同时管理多种资源，由多个计数信号量组成的一个集合称为计数信号量集，每个计数信号量管理一种资源。比如第一种资源的总数是5，第二种资源的总数是10。在使用过程中可选择申请哪种资源或哪几种资源。

坦率来讲，System V信号量有点设计过度，第二种扩展并无必要，同时操作集合中的多个信号量的能力是多余的，而这种扩展导致了编程接口过于复杂，使用不便。

10.3.2 创建或打开信号量

创建或打开信号量的函数为semget，其接口定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

这个接口比较简单，第二个参数nsems表示信号量集中信号量的个数。换句话说，就是要控制几种资源。大部分情况下只控制一种。如果并非创建信号量，仅仅是访问已经存在的信号量集，可以将nsems指定为0。

semflg支持多种标志位。目前支持IPC_CREAT和IPC_EXCL标志位，其含义不再赘述。

在创建信号量时，需要考虑的问题是系统限制。系统的限制可以分成三个层面。

- 系统容许的信号量集的上限：SEMMNI
- 单个信号量集中信号量的上限：SEMMSL
- 系统容许的信号量的上限：SEMMNS

首先介绍下对于每种限制，系统提供的硬上限，如表10-9所示。

表10-9 信号量的系统硬上限

限 制	说 明	最 大 值
SEMMNI	系统容许创建的信号量集的上限	32768 (IPCMNI)
SEMMSL	一个信号量集里信号量的最大数量	65536
SEMMNS	系统中所有信号量集里的信号量总数的上限	2147483647 (INT_MAX)

其中SEMMSL的硬上限是65536，原因是semop函数中定义了sembuf结构体来操作信号量集中的信号量，代码如下所示：

```
struct sembuf{
    unsigned short sem_num;
```

sembuf结构体中的成员变量sem_num用来指定修改集合中的哪个信号量。其数据类型是无符号短整型（unsigned short）。我们固然可以一意孤行地将SEMMSL的值设置为大于65536的数值，但是后续将无法通过semop来操作它，因此它也就失去了存在的意义。因此集合中信号量个数的硬上限值为65536。

之所以SEMMNS的上限值为INT_MAX，原因是内核使用了int型来存储该值，代码如下所示：

```
struct ipc_namespace{
    ...int sem_ctls[4];...
}
#define sc_semmsl sem_ctls[0]
#define sc_semmns sem_ctls[1]
#define sc_semopm sem_ctls[2]
#define sc_semmni sem_ctls[3]
```

在硬上限范围内，可以通过sysctl来设置软上限。

```
cat /proc/sys/kernel/sem
32000 1024000000 500 32000
sysctl kernel.sem
kernel.sem = 32000 1024000000 500 32000
```

其中4个值的含义如图10-7所示。

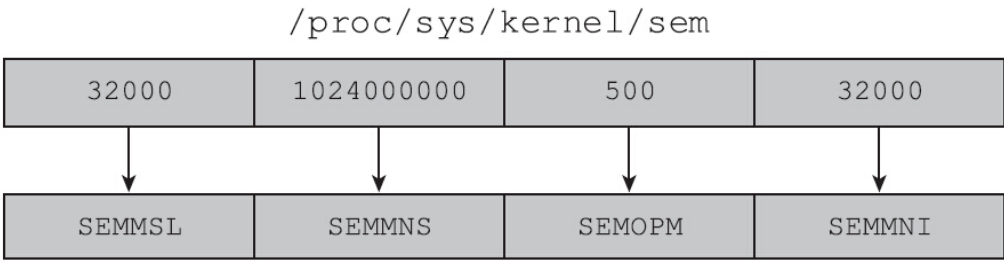


图10-7 信号量相关的控制选项的含义

第三个值（SEMOPM）的含义将放到后面再介绍。可以通过sysctl-w或修改/etc/sysctl.conf来设置控制参数。注意不要超过硬上限。

如果超过系统限制时，返回的错误码见表10-10。

表10-10 信号量超过系统限制相关的错误码

情 形	相 关 限 制	errno
因信号量集的个数达到上限而失败	SEMMNI	ENOSPC
因信号量的个数达到上限而失败	SEMMNS	ENOSPC
因单个信号量集中信号量的个数超过上限而失败	SEMMSL	EINVAL

在System V信号量的接口设计中，存在一个致命的缺陷，即创建信号量集和初始化集合中的信号量是两个独立的操作，而非一个原子操作，标准并未要求创建信号量集时，将信号量的值初始化为0。当然，在Linux系统上，semget函数返回的信号量实际上会被初始化为0。

但是很多情况下，信号量的初始值并不希望为0，因此需要额外调用一次semctl的SETVAL命令来设置初始值。由于创建和初始化之间存在一个时间窗口，因此可能会出现竞态条件（race condition），见表10-11。

表10-11 创建和初始化分开，产生竞态条件的一种情况

进 程 1	进 程 2
调用 semget 函数创建信号量集	调用 semget 函数获取信号量集的标识符 ID
	调用 semop 修改信号量的值
调用 semctl 的 SETVAL 命令初始化	

在表10-11这种时序条件下，信号量的值尚未初始化就被进程2通过semop函数修改了。而后面进程1的初始化命令又会覆盖进程2所做的更改。

W.Richard Stevens在名著《Unix网络编程卷2：进程间通信》中给出了如下思路来解决这个困境。

内核与信号量集相关的数据结构sem_array中有一个成员变量sem_otime，如下所示：

```
struct sem_array {
    ...
    time_t      sem_otime; /* 上次执行

semop的时间

*/
    ...
};
```

信号量集被创建的时候，sem_otime被初始化成0，在后续执行semop操作的时候，才会对sem_otime的值进行修改。因此可以利用这个属性来消除竞争。即第二个进程要等到创建信号量的进程执行过一次修改信号量值的semop操作后（通过判断sem_otime的值是否为0），才开始正常的流程。

《Linux/Unix系统编程手册（下册）》中也采用了这个思路解决了竞争问题，并给出了示例代码。但其示例代码适用范围比较狭窄，只适用于将信号量初始化为0这种场景。稍加改造，就可以适用于将信号量初始化为任意值的场景。具体做法如图10-8所示。

POSIX信号量作为后来者，注意到了System V信号量的这个弊端，于是将创建和初始化由一个接口来完成。详情可阅读第11章。

10.3.3 操作信号量

semop函数负责修改集合中一个或多个信号量的值，其定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

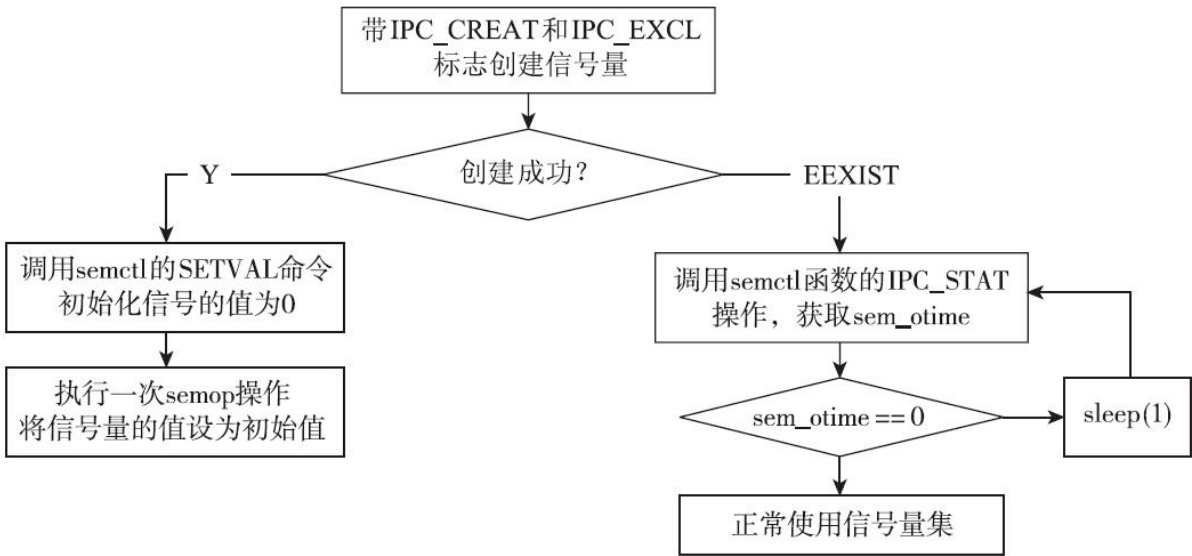


图10-8 信号量的创建和初始化流程图

函数的第一个参数是通过semget获取到的信号量的标识符ID。第二个参数是sembuf类型的指针。sembuf结构体定义在sys/sem.h头文件中。一般来说，该结构体至少包含以下三个成员变量：

```
struct sembuf {
    unsigned short int sem_num ;
    short sem_op ;
    short sem_flg;
```

成员变量sem_num解决的是操作哪个信号量的问题。因为信号量集中可能存在多个信号量，需要用这个参数来告知semop函数要操作的是哪个信号量，0表示第一个信号量，1表示第二个信号量，依此类推，最大为nsems-1，即不得超过集合中信号量的个数。如果sem_num的值小于0，或者大于等于集合中信号量的个数，semop调用则会返回失败，并置errno为EFBIG。

一般来讲，不建议采用如下方法来初始化sembuf:

```
struct sembuf myopsbuf = {1,-1,0}
```

因为考虑到可移植性，我们并没有十足的把握可以确定sembuf结构体中成员变量的顺序和上面定义中给出的顺序是严格一致的。（不过Linux的定义就是上面给出的定义，若不考虑可移植性，可以放心采用上面的方法。）

semop函数的典型用法如下所示：

```
struct sembuf myopsbuf[3] :

myopsbuf[0].sem_num = 0;    /*操作信号量集中的第

0个信号量

*/
myopsbuf[0].sem_op = -1;    /*信号量

0的值减去
```


1, 即申请

1个资源

```
*/
myopsbuf[0].sem_flg = 0 ;
myopsbuf[1].sem_num = 1;    /*操作信号量集中的第
```

1个信号量

```
*/
myopsbuf[1].sem_op = 2 ;    /*信号量
```

1的值加上

```
2*/
myopsbuf[1].sem_flg = 0;
myopsbuf[2].sem_num = 2;    /*操作信号量集中的第
```

2个信号量

```
*/
myopsbuf[2].sem_op = 0;     /*等待第
```

2个信号量的值变为

```
0*/
myopsbuf[2].sem_flg = 0;
if(semop(semid,myopsbuf,3) == -1)
{
    /*error handler here*/
}
```

`semop`函数每次会操作一组信号量，每个信号量由一个`sembuf`来表示，修改一个信号量最好也将其定义成`struct sembuf ops[1]`这样的数组，`semop`函数的第三个参数表示要操作的信号量的个数。

如果调用`semop`函数同时操作多个信号量，要被原子地执行，要么内核完成所有操作，要么内核什么也不做。

尽管信号量集支持同时操作多个信号量，但事实上这种场景是非常罕见的。大多数情况下，只会操作集合中的一个信号量。更常见的是使用如下方式。

```
struct sembuf myopsbuf[1] ;
```

```
myopsbuf[0].sem_num = 0;
myopsbuf[0].sem_op = -1;    /*信号量
```

0的值减去

```
1*/
myopsbuf[0].sem_flg = 0 ;
if(semop(semid,myopsbuf,1) == -1)
```

`sembuf`中的`sem_op`可以是正值，也可以是负值，还可以是0。介绍其含义之前，首先来介绍几个相关的变量。

·`semval`: 信号量的当前值，表示当前可用的资源个数，永远非负。

·`semzcnt`: 正在等待信号量的值变成0的进程个数。

·`semncnt`: 正在等待信号量的值大于当前值的进程个数。

根据`sem_op`的值和`sem_flg`值，`semop`函数的行为模式如表10-12所示。

表10-12 semop的含义

sem_op	含 义
>0	用于释放资源。 将对应信号量的 semval 的值加上 sem_op 的值。如有其他进程等待资源且增加后的信号量值满足其要求，则将其唤醒。
=0	用于等待信号量值 semval 变成 0。 如果当前 semval 等于 0，则立即返回成功。 semval 不等于 0 的时候： ❑ 如果指定了 IPC_NOWAIT，则立即返回失败，errno 为 EAGAIN。

(续)

sem_op	含 义
	❑ 如果未指定 IPC_NOWAIT，则陷入阻塞。semzcnt 的值加 1。 一般有三种情况可以从阻塞中醒来： ○ 若信号量值变成了 0，则成功返回，semzcnt 的个数减少 1 个。 ○ 若信号量被删除，则返回失败，并置 errno 为 ERMID。 ○ 若 semop 系统调用被信号中断，则返回失败，并置 errno 为 EINTR，semzcnt 的个数减少 1 个。
<0	用于申请资源。 若信号量的值不小于 sem_op 的绝对值，则表示资源足够可用，信号量的值 semval 减掉 sem_op 的绝对值，并成功返回。 信号量的值小于 sem_op 的绝对值时： ❑ 如果指定了 IPC_NOWAIT，则立即返回失败，errno 为 EAGAIN。 ❑ 如果未指定 IPC_NOWAIT，则陷入阻塞，semmcnt 的值加 1。 一般有三种情况可以从阻塞中醒来： ○ 若信号量的值变成了不小于 sem_op 的绝对值，则成功返回。 ○ 若信号量被删除，则返回失败，errno 为 ERMID。 ○ 若 semop 系统调用被信号中断，则返回失败，并置 errno 为 EINTR。

对于semop操作，也存在如下系统限制：

- 单次semop调用能够操作的信号量的最大值：SEMOPM
- 信号量值的上限：SEMVMX

单次semop调用能够操作的信号量的最大个数记录在procfs中：

```
sysctl kernel.sem
kernel.sem = 32000 1024000000 500 32000
```

如果nsops的值超过了SEMOPM，则semop函数返回-1，并置errno为E2BIG。

除此之外，信号量的值也是有上限的，最大值为32767。若semop的增加操作导致信号量的值超过了其上限SEMVMX，那么semop函数返回-1，并置errno为ERANGE。

通过上面的讨论，不难看出semop接口复杂难用。成熟的项目都会将semop函数封装起来，提供更好用、语义更简单的接口。对于编程者而言，不外乎申请资源（wait）和释放资源（post），可将接口进行如下封装：

```
int semaphore_wait (int semid, int index)
{struct sembuf operations[1];operations[0].sem_num = index;operations[0].sem_op = -1;operations[0].sem_flg = SEM_UNDO;return semop (semid, operations, 1);
}
int semaphore_post (int semid, int index)
{struct sembuf operations[1];operations[0].sem_num = index;operations[0].sem_op = 1;operations[0].sem_flg = SEM_UNDO;return semop (semid, operations, 1);
}
```

正常使用时，如果需要等待资源，就调用semaphore_wait函数：

```
semaphore_wait(semid, 0)
```

释放资源的时候，就调用semaphore_post函数：

```
semaphore_post(semid, 0)
```

注意，上面的封装仅仅是做一个简单的示意，很多问题并未考虑（比如未考虑系统调用被信号中断，收到EINTR错误码的场景），这些封装在项目中一般作为底层基础库，真正封装的时候要小心谨慎，考虑各种场景。

封装示例中使用了SEM_UNDO标志位，其含义见10.3.4节。

10.3.4 信号量撤销值

使用信号量存在这样一种风险，即进程申请了资源，修改了信号量的值，却还没来得及释放资源就异常退出了。异常退出的进程把资源带进了坟墓，而其他进程却在苦苦等待其释放资源。这就意味着资源泄漏，即该进程申请的资源再也无法给其他进程使用了。对于二值信号量来说，资源泄漏的危害尤其大。

为了避免因这个问题而陷入不可收拾的境地，内核提供了一种解决方案，即内核会负责记住进程对信号量施加的影响，当进程退出的时候，内核负责撤销该进程对信号量施加的影响。

调用semop函数时，可以通过如下方法设置SEM_UNDO标志位。

```
struct sembuf myopsbuf[1];
myopsbuf[0].sem_num = 0;
myopsbuf[0].sem_op = -1; /*信号量

0的值减去

1*/
myopsbuf[0].sem_flg |= SEM_UNDO ;
semop (semid,myopsbuf,1);
```

内核并不会为所有带SEM_UNDO标志位的semop操作都保存一笔记录，内核维护了一个名为semadj的变量，该变量记录了一个进程在信号量上使用SEM_UNDO操作所做的调整总和。

带SEM_UNDO标志位的semop对semadj的影响如表10-13所示。

表10-13 semadj与sem_op的关系

sem_op	semadj 的调整
>0	semadj=semadj-sem_op
<0	semadj=semadj+abs(sem_op)

当进程退出时，会将信号量的当前值加上这个semadj，来撤销进程对信号量的影响。

申请资源和释放资源时，SEM_UNDO标志位要成对地出现。切不可只在申请资源的时候使用SEM_UNDO，或者只在释放资源的时候使用SEM_UNDO，这都会造成semadj失准，不能正确地反映进程对信号量施加的影响。

当使用semctl的SETVAL或SETALL命令重新设置信号量的值时，所有使用这个信号量的进程中的semadj值都会被重置为0。因为SETVAL或SETALL相当于开启了上帝模式，强行将信号量的值设定为某个值了。

SEM_UNDO也不是包治百病的良药。信号量是用来管理资源的，本身并无实际含义，如果进程异常退出，而资源并没有进入一个合理且稳定的状态，单单调整信号量的值并不一定能使应用恢复到一个稳定一致的状态。

除此以外，在某些情况下，进程终止时，也无法严格地按照进程的semadj来调整信号量的值，考虑如下情景：

- 1) 信号量的初始值是0。
- 2) A进程将信号量增加2，并且设置了SEM_UNDO标志位。
- 3) B进程将信号量减去1，此时信号量的值变为1。
- 4) A进程退出。

按照逻辑，应该将当前信号量的值减去2。但是由于当前信号量的值是1，不可能减去2，那该怎么办呢。对于此困境，Linux采用的办法是尽可能地减小信号量的值。对于本例，就是将信号量的值减少为0。

上面的情况是向下溢出，与之对应的情况是向上溢出。即如果加上撤销量，信号量的值超过了上限SEMVMX，内核会将信号量的值调整为

SEMVMX。这部分逻辑体现在ipc/sem.c中的exit_sem函数中：

```
for (i = 0; i < sma->sem_nsems; i++) {
    struct sem * semaphore = &sma->sem_base[i];
    if (un->semadj[i]) {
        /*信号量的值加上退出进程的对应的撤销值

    */
        semaphore->semval += un->semadj[i];/*向下溢出，则置为

    0*/
        if (semaphore->semval < 0)
            semaphore->semval = 0;
        /*向上溢出，则置为

    SEMVMX*/
        if (semaphore->semval > SEMVMX)
            semaphore->semval = SEMVMX;
        semaphore->sempid = task_tgid_vnr(current);
    }
}
```

一般来讲，SEM_UNDO标志位多用于二值信号量。

10.3.5 控制信号量

控制信号量的函数为`semctl`函数，其定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, /* union semun arg*/);
```

某些特定的操作需要第四个参数，第四个参数是联合体，很不幸的是这个联合体需要程序员自己定义，代码如下所示：

```
union semun {
    int                val;
    struct semid_ds    *buf;
    unsigned short     *array;
    struct seminfo     *__buf; /*Linux特有的

    */
};
```

根据第三个参数`cmd`值的不同，`semctl`支持以下命令：

1. IPC_RMID

`semctl`函数的第二个参数被忽略。和消息队列的删除一样，内核不会维护信号量集的引用计数，说删就删，而且是立即删除信号量集。所有阻塞在`semop`函数上的进程将被唤醒，返回错误并置`errno`为`ERMID`。

删除信号量的示例代码如下：

```
int semaphore_destroy(int semid)
{
    union semun ignored_argument; semctl(semid, 0, IPC_RMID, ignored_argument);
}
```

2. IPC_STAT

用于获取信号量集的信息，并存放在`union semun`中`buf`指向的结构体。

每个信号量集都有一个与之关联的`semid_ds`结构体（该结构体无须自己定义），它至少包含以下成员：

```
struct ipc_perm sem_perm;
time_t sem_otime;
time_t sem_ctime;
unsigned long sem_nsems;
```

可以使用如下的简单代码来获取上述信息（省略错误处理）：

```
struct semid_ds ds ;
union semun arg;    /*须确保

semun联合体已经定义

*/
arg.buf = &ds ;
semctl(semid,0,IPC_STAT,arg);
printf("

last op time is %s\n"

,ctime(&(ds.sem_otime)));
```

3.IPC_SET

union semun arg的成员变量buf，可用来设置sem_perm.uid、sem_perm.gid和sem_perm.mode。

4.GETVAL

返回集合中第semnum个信号量的值，无需第四个参数，示例代码如下：

```
int semaphore_getval(int semid,int index)
{
    union semun ignored_argument;
    return semctl(semid, index, GETVAL, ignored_argument);
}
```

5.SETVAL

将信号量集中的第semnum个信号的值设置为arg.val，示例代码如下：

```
int semaphore_setval(int semid, int index, int value)
{
    union semun arg;
    arg.val = value;return semctl(semid, index, SETVAL,arg);
}
```

6.GETALL

将信号量集中所有信号的值存放在第四个参数arg的成员变量array中。确保有足够的空间可以存放array数组。这个操作将忽略第二个参数semnum。

7.SETALL

用第四个参数arg的成员变量array数组中的值初始化信号量集中的所有信号量。一般来说这个操作用于信号量的初始化，正常使用期间很少会调用SETALL。

需要注意的是如果调用了SETVAL或SETALL，使用信号量的所有进程的semadj都会被清零。

8.GETPID

返回上一个对第semnum个信号量执行semop的进程的进程ID，如果不存在，则返回0。

9.GETNCNT

返回等待第semnum个信号量值增大的进程的个数。

10.GETZCNT

返回等待第semnum个信号量值变成0的进程的个数。

10.4 System V共享内存

10.4.1 共享内存概述

共享内存是所有IPC手段中最快的一种。它之所以快是因为共享内存一旦映射到进程的地址空间，进程之间数据的传递就不须要涉及内核了。

回顾一下前面已经讨论过的管道、FIFO和消息队列，任意两个进程之间想要交换信息，都必须通过内核，内核在其中发挥了中转站的作用：

- 发送信息的一方，通过系统调用（write或msgsnd）将信息从用户层拷贝到内核层，由内核暂存这部分信息。

- 提取信息的一方，通过系统调用（read或msgrcv）将信息从内核层提取到应用层。

上述情景如图10-9所示。

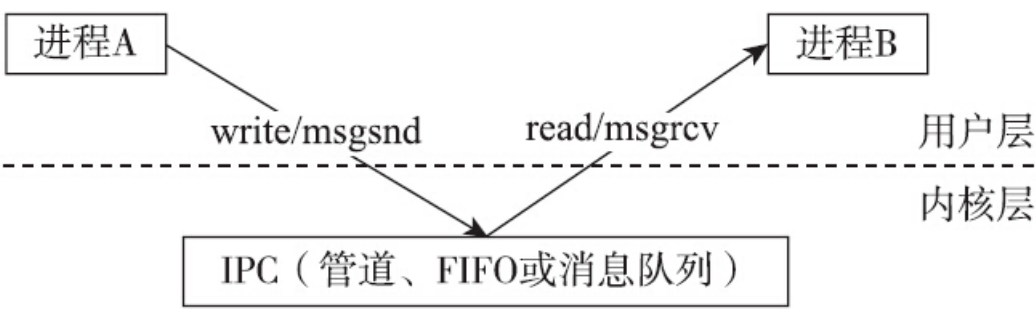


图10-9 管道、FIFO和消息队列应用层与内核的交互

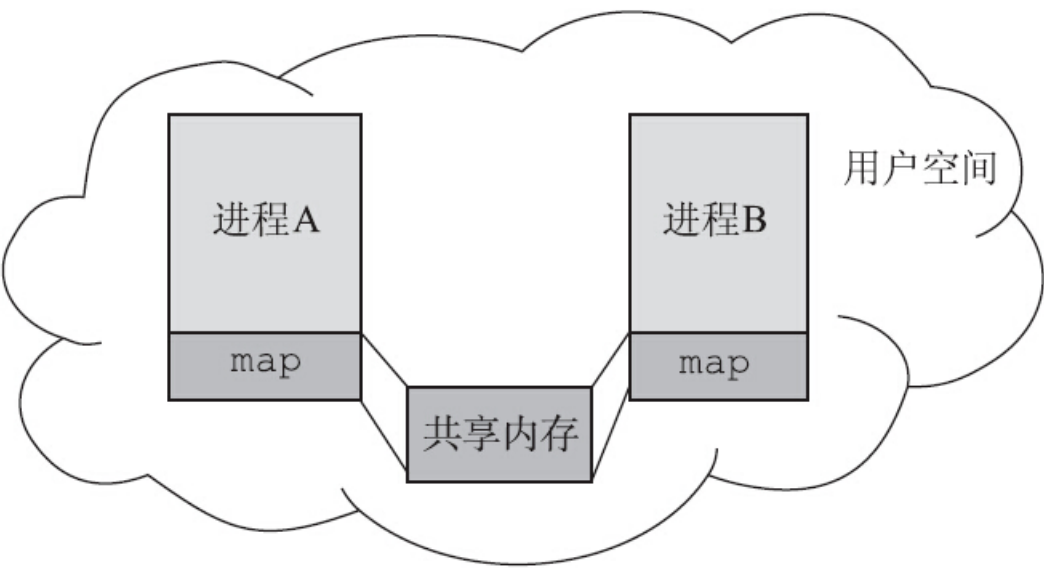


图10-10 共享内存的思想

一个通信周期内，上述过程至少牵扯到两次内存拷贝（从用户拷贝到内核空间和从内核空间拷贝到用户空间）和两次系统调用，这其中的开销不容小觑。用户层的体验固然不佳，内核层想必也是不堪其扰，双方的内心都是崩溃的。

于是，不堪其扰的内核提出了一个新的思路：共享内存，这种思路可以通俗地概括为内核搭台，进程唱戏。简单地说，内核负责构建出一片内存区域，两个或多个进程可以将这块内存区域映射到自己的虚拟地址空间，从此之后内核不再参与双方通信。正所谓：

事了拂衣去，深藏身与名。

——李白《侠客行》

进程之间使用共享内存通信的方式如图10-10所示。



注意 建立共享内存之后，内核完全不参与进程间的通信，这种说法严格来讲并不是正确的。因为当进程使用共享内存时，可能会发生缺页，引发缺页中断，这种情况下，内核还是会参与进来的。

进程从此就像操作普通进程的地址空间一样操作这块共享内存，一个进程可以将信息写入这片内存区域，而另一个进程也可以看到共享内存里面的信息，从而达到通信的目的。

允许多个进程同时操作共享内存，就不得不防范竞争条件的出现，比如有两个进程同时执行更新操作，或者一个进程在执行读取操作时，另外一个进程正在执行更新操作。因此，共享内存这种进程间通信的手段通常不会单独出现，总是和信号量、文件锁等同步的手段配合使用。

10.4.2 创建或打开共享内存

`shmget`函数负责创建或打开共享内存段，其接口定义如下：

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

其中第二个参数`size`必须是正整数，表示要创建的共享内存的大小。内核以页面大小的整数倍来分配共享内存，因此，实际`size`会被向上取整为页面大小的整数倍。

第三个参数支持`IPC_CREAT`和`IPC_EXCL`标志位。如果没有设置`IPC_CREAT`标志位，那么第二个参数`size`对共享内存段并无实际意义，但是必须小于或等于共享内存的大小，否则会有`EINVAL`错误。

和消息队列及信号量一样，对于创建共享内存，系统也存在一些限制。

- SHMMNI**：系统所能够创建的共享内存的最大个数。
- SHMMIN**：一个共享内存段的最小字节数。
- SHMMAX**：一个共享内存段的最大字节数。
- SHMALL**：系统中共享内存的分页总数。
- SHMSEG**：一个进程允许`attach`的共享内存段的最大个数。

系统允许创建的共享内存的最大个数`SHMMNI`的硬上限为`IPCMNI`（32768），软上限记录在`proc`文件系统的如下位置。

```
cat /proc/sys/kernel/shmmni
4096
```

单个共享内存段的最小字节数`SHMMIN`是1，内核并没有提供控制选项来修改这个值。实际上共享内存会向上取整到页面大小，即共享内存占用的内存总是页面大小的整数倍，因此，实际的限制为4096字节。

单个共享内存段的最大字节数为`SHMMAX`。这个值默认是32MB，可以从`procfs`中读出该限制。但是内核并没有设置硬上限。

```
cat /proc/sys/kernel/shmmax
33554432
```

很明显，32MB对某些大型的应用来说是不够用的。最典型的的就是PostgreSQL数据库。PostgreSQL数据库会征用大量的共享内存作为其内部使用的shared_buffer。因此须要修改该参数，方法为修改/etc/sysctl.conf，新增如下内容，并执行sysctl-p来重新加载。

```
kernel.shmmax = 2147483648
```

SHMALL是一个系统级别的限制，单位是页面。内核也没有提供硬上限，一般默认值为2097152，2MB个页面即 $2\text{MB} \times 4096 = 8\text{GB}$ 。该限制记录在procfs的如下位置。

```
cat /proc/sys/kernel/shmall  
2097152
```

SHMSEG是一个进程级别的限制，限制一个进程最多可以attach多少个共享内存段。内核事实上并没有特别的限制，因此该限制实际上和SHMMNI的值一样。

10.4.3 使用共享内存

`shmget`函数，不过是在茫茫内存中创建了或找到了一块共享内存区域，但是这块内存和进程尚没有任何关系。要想使用该共享内存，必须先把共享内存引入进程的地址空间，这就是`attach`操作。

`attach`操作的接口定义如下：

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

其中，第二个参数是用来指定将共享内存放到虚拟地址空间的什么位置的。大部分的普通青年都会将第二个参数设置为`NULL`，表示用户并不在意，一切交由内核做主。

当`shmaddr`的地址不是`NULL`的时候，表示进程希望将共享内存`attach`到该地址。但是该地址必须是系统分页的整数倍，否则会返回`EINVAL`错误。内核提供了一个`shmflg`为`SHM_RND`，表示该地址不是系统分页的整数倍也没关系，系统会在用户给出的地址附近，就近找一个系统分页整数倍的地址。

如果指定的`shmaddr`落在已经在用的地址范围内，就会导致`EINVAL`错误。但是Linux提供了一个非标准的扩展`SHM_REMAP`。这个标志位表示替换位于`shmaddr`处且长度为共享内存段的长度的任何内存映射。很明显，设置了`SHM_REMAP`标志位，`shmaddr`参数就不能再为`NULL`了。

如果进程仅仅是读取共享内存段的内容，并不修改，则可以指定`SHM_RDONLY`标志位。

`shmat`如果调用成功，则返回进程虚拟地址空间内的一个地址。如果失败，就会返回`(void*) -1`，并且设置`errno`。

如何通过`shmat`返回的地址来使用共享内存？答案是像使用`malloc`分配的空间一样使用共享内存。我们都使用过`malloc`，调用`malloc`时，会指定分配空间的大小，`malloc`成功后，可以正常地使用返回的地址（只要不超过分配的空间）。`shmat`也是一样，程序员可以自如地使用`shmat`返回的地址。

使用共享内存和使用`malloc`分配的空间还是有区别的。共享内存段用于多个进程间的通信，因此，写入共享内存的内容要事先约定好，读取进程才可以正常地解析写入进程写入的内容。`malloc`分配的内存区域完全归调用进程所有，其他进程不可见，但共享内存则不然，其他进程也可能会同时操作该共享内存，因此使用者要有进程间同步的觉悟。

下面给出一个将共享内存`attach`到进程地址空间的例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <errno.h>
#include <string.h>
#define MYKEY 0x3333
int main()
{
    int shmid;
    void *ptr = NULL;
    shmid = shmget(MYKEY, 4096, IPC_CREAT | IPC_EXCL | 0640);
    if (shmid == -1)
    {
        if (errno != EEXIST)
        {
```

```

        fprintf(stderr, "shmget returned %d (%d: %s)\n",
                shmkey, errno, strerror(errno));
        return 1;
    }
    else
    {
        shmkey = shmget(MYKEY, 4096, 0);
        if (shmkey == -1)
        {
            fprintf(stderr, "shmget returned %d (%d: %s)\n",
                    shmkey, errno, strerror(errno));
            return 2;
        }
    }
}
fprintf(stdout, "shmkey = %d\n", shmkey);
ptr = shmat(shmkey, NULL, SHM_RDONLY);
if (ptr == (void*)-1)
{
    fprintf(stderr, "shmat return NULL, errno (%d: %s)\n",
            errno, strerror(errno));
    return 2;
}
fprintf(stdout, "shmat returned %p\n", ptr);
sleep(1000);
shmdt(ptr);
return 0;
}

```

当执行上述程序时，可以看到如下输出：

```

./shm
shmkey = 131075
shmat returned 0x7f555dc5c000

```

可以看到返回的标识符ID为131075，该共享内存attach到进程的地址空间后，在进程内的地址为0x7f555dc5c000。

通过查看进程的地址空间，也可以看出共享内存所在的位置，代码如下：

```
cat /proc/9058/maps...
```

```

7f34c6de8000-7f34c6de9000 rw-s 00000000 00:04 131075
/SYSV00003333 (deleted)

```

上述输出中，字段的含义如图10-11所示。

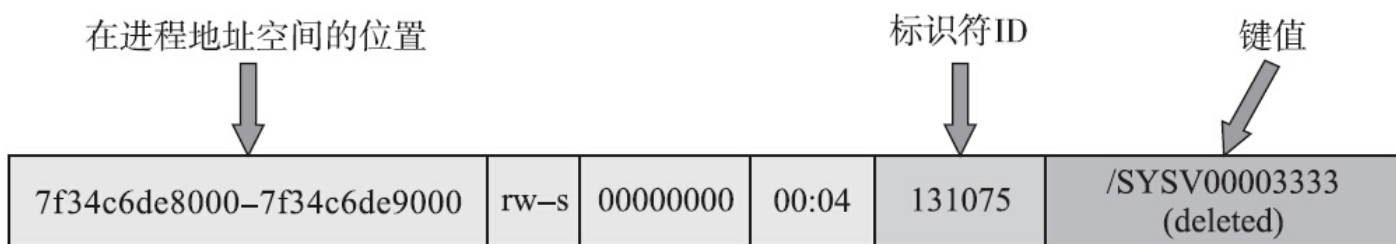


图10-11 /proc/PID/maps中的共享内存段

共享内存和System V消息队列及System V信号量有不同之处，共享内存维护了attach该共享内存的进程的个数，见下面输出的nattach列：

```

ipcs -m
----- Shared Memory Segments -----
key      shmkey  owner    perms    bytes    nattach    status
0x07021999 196608  root     644     1712     2          ...

```

存在引用计数，就不难猜出共享内存的删除和消息队列及信号量的删除是不同的。它并不遵循说删就删的准则，删除时会判断**attach**该共享内存的进程个数。如果尚有进程在使用该共享内存，就不会真正地删除，而是让内核负责标记一下就返回了。

正是因为**attach**操作会影响删除的行为，因此，使用共享内存的进程如果确认不再使用了，应该及时地将共享内存分离，使其离开进程的地址空间，这就是分离操作。分离会使共享内存的引用计数减1。

通过**fork**函数创建的子进程，会继承父进程**attach**的共享内存。因此在**fork**之前创建共享内存，后面父子进程就可以使用这块共享内存进行通信了。

10.4.4 分离共享内存

分离操作的接口定义如下：

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

`shmdt`函数仅仅是使进程和共享内存脱离关系，并未删除共享内存。`shmdt`函数的作用是将共享内存的引用计数减1。如前所述，只有共享内存的引用计数为0时，调用`shmctl`函数的`IPC_RMID`命令才会真正地删除共享内存。

进程执行`exec`之后，所有`attach`的共享内存都会被分离。当进程终止之后，共享内存也会自动被分离。

10.4.5 控制共享内存

shmctl函数用来控制共享内存，函数接口定义如下：

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

当cmd为IPC_STAT和IPC_SET时，需要用到第三个参数。其中shmid_ds结构体的定义如下：

```
struct shmid_ds {
    struct ipc_perm shm_perm;
    size_t          shm_segsz;
    time_t          shm_atime;
    time_t          shm_dtime;
    time_t          shm_ctime;
    pid_t           shm_cpid;
    pid_t           shm_lpid;
    shmatt_t        shm_nattch;
    ...
};
```

1.IPC_STAT

用于获取shmid对应的共享内存的信息。所谓信息，就是上面结构体的内容。

shm_perm中的mode字段有两个比较特殊的标志位，即SHM_DEST和SHM_LOCKED。

删除共享内存时，可能由于attach它的进程个数不为0，因此只能打上一个标记，表示标记删除，待到所有attach该共享内存的进程都执行过分离（detach）操作，共享内存的引用计数变成0之后，才执行真正的删除操作。所谓的标记指的就是SHM_DEST标志位。

对于已经标记删除的共享内存，可以通过ipcs-m命令的status栏来查看，其dest含义是已经标记删除的意思。

key	shmid	owner	perms	bytes	nattch	status
0x00000000	32768	root	666	4096	1	dest

可以通过shmctl的SHM_LOCK操作将一个共享内存段锁入内存，这样它就不会被置换出去。这样做的好处是访问共享内存的时候，不会产生缺页中断（page fault）。

通过ipcs-m的输出可以查看共享内存是否被锁入内存，注意下面状态中的locked字段，该字段表明对应的共享内存已被锁入内存。

```
ipcs -m
----- Shared Memory Segments -----
key      shmid  owner   perms  bytes  nattch  status
0x00003333 32768  manu   640    4096    1       locked
```

除此以外，其他字段就顾名思义了。

·shm_segsz: 共享内存的字节数。

·shm_atime: 创建共享内存时设置成0，当进程通过shmat函数attach共享内存时，将时间更新为当前时间。

·shm_dtime: 创建共享内存时设置成0，当进程调用shmdt分离共享内存时，将时间更新成当前时间。

·shm_ctime: 当创建共享内存时，设置该值为当前时间；当调用IPC_SET操作时，更新该值为当前时间。

·shm_nattch: attach该共享内存到其地址空间的进程的个数。

2.IPC_SET

IPC_SET也只能修改shm_perm中的uid、gid及mode。

3.IPC_RMID

可以通过如下方式删除共享内存段：

```
ret = shmctl(shmid, IPC_RMID, (struct shmid_ds *) NULL);
```

如果共享内存的引用计数shm_nattch等于0，则可以立即删除共享内存。但是如果仍然存在进程attach该共享内存，则并不执行真正的删除操作，而仅仅是设置SHM_DEST标记。待所有进程都执行过分离操作之后，再执行真正的删除操作。

值得一提的是，共享内存处于SHM_DEST状态的情况下，依然允许新的进程调用shmat函数来attach该共享内存。

4.SHM_LOCK

可以通过如下方式将共享内存锁定在内存之中：

```
ret = shmctl(shmid, SHM_LOCK, (struct shmid_ds *) NULL);
```

上面的代码会将共享内存锁定在RAM中，而不被置换出去。这种做法可以提升共享内存的访问性能。因为进程在访问共享内存所在的分页时，不会因缺页中断而导致性能下降。

注意调用SHM_LOCK并不能保证在shmctl函数结束时，所有的共享内存页已经位于RAM中了，当没有驻留在RAM中的页面因为访问需要，由缺页中断而被引入RAM后，该页面就会被锁定，而不会被交换出去。除非调用了下面提到的SHM_UNLOCK，否则页面会一直驻留在内存中。

SHM_LOCK设置的是共享内存的属性，而不是进程的属性，所以哪怕所有attach共享内存的进程都已终止，共享内存的页面仍被锁定在RAM中。故而为了防止发生资源泄漏，要及时解锁已锁定的共享内存。解锁操作可通过shmctl函数的SHM_UNLOCK来完成。

5.SHM_UNLOCK

SHM_UNLOCK操作和SHM_LOCK操作相反，是解锁操作，即允许共享内存的页面被交换出去。可以通过如下方式解锁共享内存：

```
ret = shmctl(shmid, SHM_UNLOCK, (struct shmid_ds *) NULL);
```

第11章 进程间通信：POSIX IPC

与System V IPC一样，POSIX IPC也包含三种类型：

- POSIX消息队列
- POSIX信号量（又分为命名信号量 and 无名信号量）
- POSIX共享内存

POSIX IPC的出现要比System V IPC晚，因此POSIX IPC的设计者可以从容地参照System V IPC，吸收其设计上的长处，规避其设计上的缺点。正是由于POSIX IPC拥有后发优势，所以总体来讲，POSIX IPC要优于System V IPC。

表11-1汇总了POSIX IPC的所有函数。

表11-1 POSIX IPC函数列表

	消 息 队 列	信 号 量	共 享 内 存
头文件	<mqueue.h>	<semaphore.h>	<sys/mman.h>
创建或打开	mq_open	sem_open	shm_open+mmap
关闭	mq_close	sem_close	munmap
删除	mq_unlink	sem_unlink	shm_unlink
执行 IPC	mq_send mq_receive	sem_post sem_wait sem_getvalue	在共享内存区域内操作数据
其他操作	mq_getattr mq_setattr mq_notify	sem_init（初始化未命名信号量） sem_destroy（销毁未命名信号量）	无

11.1 POSIX IPC概述

在POSIX IPC的模型中，对open、close和unlink等类似函数（见表11-1创建或打开、关闭和删除三行）的使用与传统的Unix文件模型一致，相信理解和操作起来应该很容易。

与打开文件一样，POSIX IPC对象也有引用计数，内核会负责维护IPC对象上的打开引用计数。它所带来的影响是删除POSIX IPC对象的操作比较简单。删除操作仅仅是删除IPC对象的名字，等所有的进程都使用完毕，IPC对象的引用计数变成0之后才真正销毁IPC对象。

11.1.1 IPC对象的名字

多个进程之间操作同一个IPC对象，总要有个入口点或线索，以便根据线索找到共同的IPC对象。

对于System V IPC而言，键值就是其线索，只要拿着相同的键值就能找到同一个System V IPC对象（如图11-1所示）。

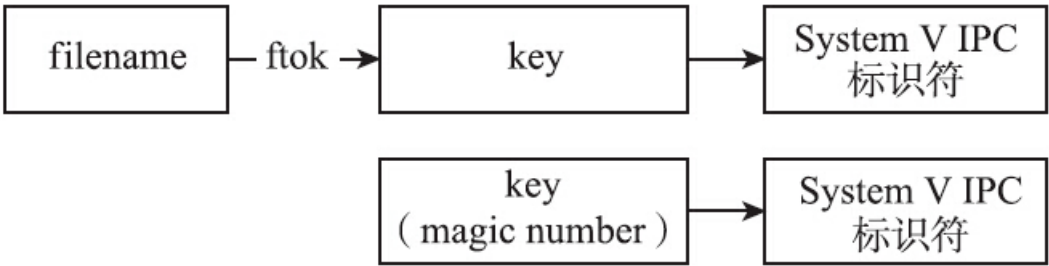


图11-1 System V IPC顺藤摸瓜之藤

对于POSIX IPC来说，可以像操作文件一样操作IPC对象。文件有路径名，同样，IPC对象也有IPC对象的名字。SUSv3标准规定，唯一一种用来标识POSIX IPC对象的可移植方法是使用以斜线打头后面跟着一个或多个非斜线字符的名字，如/myobject。

下面三段代码分别负责创建POSIX消息队列、信号量和共享内存。

```
/*创建

POSIX消息队列

*/
mqd_t mqd = mq_open(argv[1],O_RDWR|O_CREAT|O_EXCL,S_IRUSR|S_IWUSR,NULL);
if(mqd == -1)
{
    /*error handle*/
}
/*创建

POSIX信号量

*/
sem_t* sem = sem_open(argv[1],O_CREAT|O_EXCL,S_IRUSR|S_IWUSR,1);
if(sem == SEM_FAILED)
{
    /*error handle*/
}
/*创建

POSIX 共享内存
```

```
*/
int shm_fd= shm_open(argv[1],O_RDWR|O_CREAT|O_EXCL,S_IRUSR|S_IWUSR);
if (shm_fd == -1)
{
    /*error handle*/
}
```

Linux为IPC对象提供了文件系统的访问接口，即可以像操作普通文件一样操作IPC对象。

对于创建出来的共享内存和信号量，Linux将这些对象放到了挂载在/dev/shm目录处的tmpfs文件系统中，代码如下所示：

```
ll /dev/shm/
total 0
drwxrwxrwt  2 root root  40 Sep 12 05:08 ./
drwxr-xr-x 20 root root 780 Sep 12 04:22 ../创建名为
```

abc的

POSIX共享内存之后

```
./shm_open abc
ll /dev/shm/
total 0
drwxrwxrwt  2 root root  60 Sep 12 05:13 ./
drwxr-xr-x 20 root root 780 Sep 12 04:22 ../
-rw-----  1 manu manu   0 Sep 12 05:13 abc创建名为
```

abc的

POSIX信号量之后

```
./sem_open abc
ll /dev/shm/
total 4
drwxrwxrwt  2 root root  80 Sep 12 05:14 ./
drwxr-xr-x 20 root root 780 Sep 12 04:22 ../
-rw-----  1 manu manu   0 Sep 12 05:13 abc
-rw-----  1 manu manu  32 Sep 12 05:14 sem.abc
```

可以看到，创建一个名为name的共享内存后，在/dev/shm目录下就会有一个名为name的文件。如果创建一个名为name的信号量，那么在/dev/shm目录下就会有一个名为sem.name的文件。

消息队列也可以展现在文件系统中，不过要比共享内存和信号量稍微复杂一些。需要首先将消息队列挂载到文件系统中，方法如下：

```
mkdir /dev/mqueue
mount -
```

```
t mqueue none /dev/mqueue
```

现在可以创建消息队列了。当然如果不将消息队列挂载到文件系统中，并不会影响消息队列的创建，仅仅是无法从文件系统查看消息队列的情况而已。

```
ll /dev/mqueue/
total 0
drwxrwxrwt  2 root root   40 Sep 12 05:29 ./
drwxr-xr-x 17 root root 4260 Sep 12 03:57 ../创建一个名为
```

/abc的

POSIX消息队列

```
./mq_open /abc
ll /dev/mqueue/
total 0
drwxrwxrwt  2 root root   60 Sep 12 05:41 ./
drwxr-xr-x 17 root root 4260 Sep 12 03:57 ../
-rw-----  1 manu manu   80 Sep 12 05:41 abc
```

IPC对象的名字有哪些限制？通过测试不难得出以下结论：

- POSIX消息队列的名字必须以/打头，而且后续字符不允许出现/，否则就返回EINVAL错误。
- POSIX消息队列的名字中打头的/字符不计入长度。
- POSIX消息队列名字的最大长度为NAME_MAX（255个字符），若超过则返回ENAMETOOLONG错误。
- POSIX信号量和共享内存的名字可以以1个或多个/打头，也可以不以/打头。
- POSIX信号量和共享内存的名字中，打头的一个或多个/字符不计入长度。
- POSIX共享内存名字的最大长度为NAME_MAX，POSIX信号量名字的最大长度为NAME_MAX-4（因为实现会在信号量的名字前面添加sem.这4个字符）。若超过则返回ENAMETOOLONG错误。

注意，这些结论是从glibc相关函数（mq_open、sem_open和shm_open）的角度来分析的，并不是从系统调用的角度来分析的。glibc调用系统调用之前会做一些动作，比如mq_open函数调用同名系统调用前会去除打头的/等。

11.1.2 创建或打开IPC对象

解决了IPC对象的名字问题，接下来就是创建POSIX IPC对象了。创建或打开，都是由open系列函数来完成的。后续的操作要作用在open函数返回的句柄上。

对于POSIX IPC的open系列函数而言，一般至少包含三个参数name、oflag和mode（见表11-2）。

name前面已经说过，就是POSIX IPC的名字。下面来分析第二个参数打开标志位。

表11-2 POSIX IPC open中的标志位

	mq_open	sem_open	shm_open
只读	O_RDONLY		O_RDONLY
只写	O_WRONLY		
可读可写	O_RDWR		O_RDWR
若不存在则创建	O_CREAT	O_CREAT	O_CREAT
排他性创建	O_EXCL	O_EXCL	O_EXCL
非阻塞模式	O_NONBLOCK		
若已存在则截断			O_TRUNC

如果oflag中指定了O_CREAT标志位，则需要第三个参数mode来指定权限，这个权限和文件的权限一样，不外乎S_IRUSR、S_IWUSR、S_IRGRP、S_IWGRP、S_IROTH及S_IWOTH这6种权限。并且和open函数一样，mode中的权限会根据进程的umask取掩码。

打开还是创建，取决于oflag是否设置了O_CREAT及O_EXCL标志位。内在的控制逻辑和System V IPC一致，如表11-3所示。

表11-3 POSIX IPC O_CREATE和O_EXCL标志位的影响

oflag 标志位	对象不存在	对象存在
无特殊标志	出错，errno 为 ENOENT	成功，引用已存在对象
O_CREAT	成功，创建新对象	成功，引用已存在对象
O_CREAT O_EXCL	成功，创建新对象	失败，errno 为 EEXIST

11.1.3 关闭和删除IPC对象

POSIX IPC对象维护有引用计数，在用完IPC对象后，可以调用相关的close函数来释放与该对象关联的资源并使引用计数减1。对于消息队列，该函数是mq_close；对于信号量该函数是sem_close。共享内存和前两者略有不同，它通过munmap解除映射来解除和共享内存的关系。

当进程退出或执行exec系列函数时，IPC对象会自动关闭。

正是因为POSIX IPC对象有引用计数，所以删除的时候比较方便。对应的unlink操作会删除对象的名字，直到所有进程使用完毕，关闭了对象或解除了映射关系之后，才会真正销毁。

因为Linux提供了文件系统访问方式，因此完全可以在文件系统中执行ls或rm操作来查看或删除IPC对象。细心的读者可以看出存放IPC对象的目录都设置了粘滞位，这是用来保护目录下的文件的，即对于非特权进程只能删除它自己拥有的POSIX IPC对象。

```
ll /dev/shm/
total 0
drwxrwxrwt  2 root root  40 Sep 12 07:08 ./
ll /dev/mqueue/
total 0
drwxrwxrwt  2 root root   40 Sep 12 07:08 ./
```

11.1.4 其他

与System V IPC相比，POSIX有很多优势。后面介绍POSIX IPC的每一种通信手段的时候，都会与System V IPC对应的手段进行比较。但POSIX IPC也有明显的劣势——可移植性。因为System V出现得早，几乎所有的Unix平台都支持System V IPC。但是如果专注于Linux平台的话，这个问题就不存在了。2.6.6之后的内核版本，三种POSIX IPC手段就已经齐备。而主流在用的Linux版本很少有低于2.6.6的。

编译使用POSIX IPC的程序时需要注意以下两点。

- 当使用消息队列和共享内存的时候，需要和实时库librt链接起来。cc命令中需指定-lrt。
- 当使用信号量的时候，需要和线程库libpthread链接起来。cc命令中需指定-lpthread。

示例代码如下所示：

```
gcc -o mq_open mq_open.c -
```

```
lrt  
gcc -o shm_open shm_open.c -lrt  
gcc -o sem_open sem_open.c -
```

```
lpthread
```

11.2 POSIX消息队列

POSIX消息队列与System V消息队列有一定的相似之处，信息交换的基本单位是消息，但也有显著的区别。

最大的区别当属在Linux实现里POSIX消息队列的句柄本质是文件描述符。这个性质给POSIX消息队列带来了巨大的优势。因为是文件描述符，所以可以使用I/O多路复用系统调用（select、poll或epoll等）来监控这个文件描述符。

其次，POSIX消息队列提供了通知功能，当消息队列中有消息可用时，就会通知到进程。而System V消息队列没有通知功能，所以消息队列上何时消息进程无从得知，只能阻塞（msgrcv）或轮询（带IPC_NOWAIT标志位的msgrcv）。

最后，System V消息队列的消息提取要比POSIX消息队列灵活。POSIX消息队列本质是个优先级队列。而System V消息中存在类型字段，可以提取类型等于某值的消息，这点POSIX消息队列是做不到的。这个优势让System V消息队列在与POSIX消息队列的对决中，稍稍挽回一点颜面。

11.2.1 消息队列的创建、打开、关闭及删除

之所以在本节介绍三个接口，是因为POSIX消息队列的接口和操作文件的接口非常类似。

消息队列的mq_open函数如同操作文件的open函数，用于创建或打开一个消息队列，其接口定义如下：

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

oflag允许的标志位包括O_RDONLY、O_WRONLY、O_RDWR、O_CREAT、O_EXCL及O_NONBLOCK。

除了O_NONBLOCK标志位，其他都是老朋友了，不必赘述，这里单提一下O_NONBLOCK。如果打开消息队列时，没有设置O_NONBLOCK标志位，那么后续的mq_send调用和mq_receive调用就可能会陷入阻塞。反之，如果打开消息队列时设置了该标志位，发送消息或接受消息若不能立刻返回，则立刻返回失败，并置errno为EAGAIN，而不会陷入阻塞。

第三个参数mode和第四个参数attr只有在创建消息队列的时候才有意义。如果仅仅是打开消息队列，则无需这两个参数。mode设置的是访问权限，attr设置的是消息队列的属性。在介绍mq_getattr函数和mq_setattr函数时会展开说明。默认情况下，第四个参数可以传递NULL，表示创建默认属性的消息队列。

当mq_open调用成功时则返回一个mqd_t类型的消息队列描述符。对于Linux平台而言，这就是一个int型数字，其实这个数字和open函数返回的文件描述符本质上是一样的，从内核的ipc/mqueue.c中mq_open系统调用的实现就可以看出：

```
SYSCALL_DEFINE4(mq_open, const char __user *, u_name, int, oflag, mode_t, mode, struct mq_attr __user *, u_attr)
{
    ...
    fd = get_unused_fd_flags(O_CLOEXEC);
    ...
    return fd;
}
```

在/proc/PID/fd目录下，也可以看到消息队列对应的文件描述符：

```
./mq_open /abc
ll /proc/2925/fd
...
lrwx----- 1 manu manu 64 Sep 13 09:04 3 -> /abc
```

一个进程允许打开多少个消息队列？标准并没有严格限定，这点是由具体的实现来决定的。SUSv3标准要求这个限制最小为_POSIX_MQ_OPEN_MAX（8）。Linux没有定义这个限制。相反因为消息描述

符被实现成了文件描述符，因此其必须遵循文件描述符的限制。

进程允许打开的消息队列个数是否仅仅受限于进程打开的最大文件个数？事实上并非如此。资源限制中有一项RLIMIT_MSGQUEUE，用于限制用户在POSIX消息队列中可以分配的最大字节数。在下一节介绍POSIX消息队列的属性时，会重点介绍该限制对允许打开的消息队列个数的影响。

调用fork之后，子进程也获得了消息队列描述符的副本，这个副本会引用同样的打开的消息队列。

调用exec之后，由于内核实现中消息队列的描述符自动带有O_CLOEXEC标志位，所以其打开的消息队列会被自动关闭。

当进程退出时，所有打开的消息队列都会被关闭。

mq_close函数用于关闭消息队列描述符，这个函数和关闭文件的close函数十分类似：

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
```

如果进程已经注册了消息通知，那么消息通知也会被删除。因为任一时刻，只能有一个进程向特定消息队列注册并接收消息通知，因此删除消息通知后，其他进程就能注册消息通知了。

POSIX消息队列也具有内核持久性，纵然打开该消息队列的所有进程都执行了mq_close，消息队列的引用计数已变为0，但只要不显式地调用mq_unlink，该队列及队列上的消息依然存在。要销毁消息队列，需要调用mq_unlink函数，代码如下：

```
#include <mqueue.h>
int mq_unlink(const char *name);
```

下面通过两个测试程序来学习消息队列的创建。

第一个小程序是用来创建消息队列的，如果传入了-e选项，则表示创建时要加上O_EXCL标志位：

```
#include <mqueue.h>
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
int main(int argc, char *argv[])
{
    int c, flags;
    mqd_t mqd;
    flags = O_RDWR | O_CREAT;
    while ((c = getopt(argc, argv, "e")) != -1)
    {
        switch(c)
        {
            case 'e':
                flags |= O_EXCL;
                break;
        }
    }
    if (optind != argc - 1)
    {
```

```

        fprintf(stderr, "usage:mqcreate [-e] <name>\n");
        return -1;
    }
    mqd = mq_open(argv[optind], flags, S_IRUSR|S_IWUSR, NULL);
    if(mqd == -1)
    {
        fprintf(stderr, "mq_open failed (%s)\n",
strerror(errno));
        return -2;
    }
    mq_close(mqd);
    return 0;
}

```

第二个小程序是用来删除POSIX消息队列的：

```

#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        fprintf(stderr, "usage mqunlink <name>\n");
        return -1;
    }
    int ret = mq_unlink(argv[1]);
    if(ret != 0)
    {
        fprintf(stderr, "mq_unlink failed (%s)\n",
strerror(errno));
        return -2;
    }
    return 0;
}

```

Linux下POSIX提供了mqueue类型的虚拟文件系统，可以通过挂载，很方便地使用ls和rm来列出或删除POSIX消息队列。

可以通过如下命令将消息队列挂载到文件系统：

```
mount -t mqueue source target
```

其中source可以为none，target是挂载点。比如可以通过如下命令挂载消息队列：

```
mkdir /dev/mqueue
mount -t mqueue none /dev/queue
```

使用第一个程序编译出mqcreate二进制程序，使用第二个程序编译出mqunlink二进制程序，可以做如下试验：

```

./mqcreate /abcd
ll /dev/mqueue总用量

0
-rw----- 1 manu manu 80 3月

9 22:26 abcd
cat /dev/mqueue/abcd
QSIZE:0          NOTIFY:0          SIGNO:0          NOTIFY_PID:0

```

可以看出，通过mqcreate创建出来的消息队列，可以通过ls/dev/mqueue来查看，甚至可以通过cat/dev/mqueue/queue_name来获取消息队列的信息。

11.2.2 消息队列的属性

介绍mq_open函数时曾提到，第四个参数是mq_attr类型的，表示消息队列的属性。创建时可以指定消息队列的属性，POSIX消息队列也提供了mq_setattr函数来改变消息队列的属性。

在继续讨论之前，首先需要了解消息队列有哪些属性，mq_attr结构体中定义了以下成员。

```
struct mq_attr {
    long mq_flags;
    long mq_maxmsg;
    long mq_msgsize;
    long mq_curmsgs;
}
```

这个结构体定义在<mqueue.h>文件中：

·mq_flags: 0或设置了O_NONBLOCK。

·mq_maxmsg: 消息队列中的最大消息个数。

·mq_msgsize: 单条消息允许的最大字节数。

·mq_curmsgs: 消息队列当前的消息个数。

如果调用mq_open函数创建POSIX消息队列时，第四个参数为NULL，那么将使用默认属性。可以使用如下代码来获取默认属性：

```
int ret = mq_getattr(mqd, &attr);
if (ret != 0)
{
    fprintf(stderr, "failed to get attr(%d: %s)\n", errno, strerror(errno));
    return 2;
}
fprintf(stdout, "the default mq_maxmsg = %ld\nthe default mq_msgsize = %ld\n",
        attr.mq_maxmsg, attr.mq_msgsize);
```

其输出如下：

```
the default mq_maxmsg = 10
the default mq_msgsize = 8192
```

从输出可以看出，默认情况下，消息队列的最大消息数为10，单条消息的最大字节数为8192字节。

其中消息队列的最大消息数的默认值10记录在如下位置：

```
cat /proc/sys/fs/mqueue/msg_default
10
```

单条消息的最大字节数的默认值8192记录在如下位置：

```
cat /proc/sys/fs/mqueue/msgsize_default
8192
```

消息队列中只能存放10条消息，这明显太少了，此外单条消息的最大字节数8192可能也无法满足我们的需要。因此创建消息队列的时候需要定制属性，定制方法如下所示：

```
attr.mq_maxmsg = atoi(argv[2]);
attr.mq_msgsize = atoi(argv[3]);
mqd_t mqd = mq_open(argv[1], O_RDWR|O_CREAT |O_EXCL, S_IRUSR|S_IWUSR, &attr);
if (mqd == -1)
{
    fprintf(stderr, "failed to get mqueue (%d: %s)\n", errno, strerror(errno));
    return 1;
}
```

但是消息队列的最大消息数和单条消息的最大字节数并不能被随意指定。它受限于多个控制选项。

对于普通用户（非特权用户）而言，内核提供了两个控制选项：

```
cat /proc/sys/fs/mqueue/msg_max
10
cat /proc/sys/fs/mqueue/msgsize_max
8192
```

这两个值分别是最大消息数的上限和单条消息最大字节数的上限。普通用户在定制消息队列属性的时候不能超越这个上限。这两条限制是针对普通用户而言的，对于特权用户而言可以忽视这两条限制。

很明显，这个上限值并不大，特权用户可以调整这两项的值：

```
sysctl -w fs.mqueue.msg_max=4096
fs.mqueue.msg_max = 4096
sysctl -w fs.mqueue.msgsize_max=65536
fs.mqueue.msgsize_max = 65536
```

但是不能随意设置上限值，对于`/proc/sys/fs/mqueue/msg_max`，系统提供了硬上限`HARD_MSGMAX`，见表11-4。

表11-4 消息队列最大消息数的硬上限

内 核 版 本	HARD_MSGMAX
低于或等于 2.6.32	131072/sizeof (void*)
2.6.33 至 3.4	(32768 * sizeof (void *) / 4)
3.5 版本及以上	65536

对于`/proc/sys/fs/mqueue/msgsize_max`，系统也提供了硬上限，见表11-5。


表11-5 消息队列中单条消息最大字节数的硬上限

内 核 版 本	msgsize_max 系统硬上限
2.6.28 版本之前	INT_MAX
2.6.28 版本至 3.4 版本	1048576 (1MB)
3.5 版本及以上的版本	16777216 (16MB) 该值对特权进程也有效

通过调整对应的控制选项，可以让消息队列容纳更多的消息，或者让每条消息可以容纳更多的内容。

可是事实上，除了上述控制选项外，还存在其他限制。如果调整`msg_max`控制选项到4096，调整`msgsize_max`控制选项到65536字节，那么可以创建出能容纳4096条消息，每条消息的最大长度为64字节的消息队列；也可以创建出只容纳两条消息，每条消息最大长度为65536字节的消息队列。但是无法创建出既可以容纳4096条消息，每条消息的最大长度又为65536字节的消息队列。这表明除了上述两条控制外，还存在其他限制。

该限制就是介绍`mq_open`时提到的`RLIMIT_MSGQUEUE`。`RLIMIT_MSGQUEUE`属于资源限制的范畴。它限制了用户可以在POSIX消息队列中分配的最大字节数。注意不是单个消息队列的最大字节数，也不是一个进程能分配的最大字节数，而是该用户创建的所有的消息队列的最大字节数。如果新建消息队列会导致所有消息队列的字节数超出此限制，那么调用`mq_open`函数时会返回`EMFILE`错误。

 **注意** Robert Love大师在《Linux系统编程》中提到的返回`ENOMEM`是错误的。

`RLIMIT_MSGQUEUE`默认为819200字节，可以通过如下指令来查看：

```
ulimit -q
819200
```

我曾经遇到这样一个问题：当我在Ubuntu 12.04上创建第10个默认属性的POSIX消息队列时，会返回`EMFILE`错误，这说明默认情况下最多只能创建9个消息队列。下面来细细分析这个场景。

默认情况下，单个消息队列最多有10条消息，每条消息的最大字节数为8192。这就意味着该消息队列满载的时候，会占用81920字节。

按照`RLIMIT_MSGQUEUE`的含义，应该可以创建10个消息队列，可是为什么却只能创建9个默认属性的消息队列呢？

原因是消息队列消耗的空间，不能仅仅计算消息体（payload），还要考虑额外的开销。可以从内核的`mqueue_get_inode`函数中找到答案。

```
/*mq_msg_tblsz是额外的开销

*/
mq_msg_tblsz = info->attr.mq_maxmsg * sizeof(struct msg_msg *);
info->messages = kmalloc(mq_msg_tblsz, GFP_KERNEL);
if (!info->messages)
    goto out_inode;
/*mq_bytes是消息队列真正消耗的空间

*/

mq_bytes = (mq_msg_tblsz +
             (info->attr.mq_maxmsg * info->attr.mq_msgsize));
spin_lock(&mq_lock);
if (u->mq_bytes + mq_bytes < u->mq_bytes ||
    u->mq_bytes + mq_bytes > task_rlimit(p, RLIMIT_MSGQUEUE)) {
    spin_unlock(&mq_lock);
    /* mqqueue evict_inode() releases info->messages */
    ret = -EMFILE;
    goto out_inode;
}
```

从上面的代码不难看出，一个消息队列消耗的总空间为：

```
bytes = (attr.mq_msgsize + sizeof(struct msg_msg*)) * attr.mq_maxmsg
```

因此当RLIMIT_MSGQUEUE的值为819200字节时，单个用户是无法创建出10个默认属性的消息队列的。



注意 上述计算消息队列消耗空间的计算公式仅仅适用于某些内核版本，并不能当成绝对的公式。对于不同的内核版本，需要查看内核的mqqueue_get_inode函数来确定。对这个话题感兴趣的可以参阅内核开发邮件列表中的Document POSIX MQ/proc/sys/fs/mqueue files话题。链接为：<https://lkml.org/lkml/2014/9/29/116>

要想让消息队列中容纳足够多的消息，每条消息也足够大，那就需要同时修改RLIMIT_MSGQUEUE的值。可以通过ulimit命令来修改，也可以通过setrlimit函数来修改。

消息队列创建以后可以通过调用mq_setattr来修改属性，相关接口定义如下：

```
#include <mqueue.h>
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

对于mq_maxmsg和mq_msgsize这两个属性，在消息队列创建的时候，就已经确定下来了，虽然提供有mq_setattr函数，但是该函数并不能修改这两个属性。

该函数可以改变的属性只有第一个mq_flags，即可以通过改变O_NONBLOCK标志位来确定是否置位。其他的属性均不可以修改。改变O_NONBLOCK属性的方法如下：

```
mq_getattr(mqd, &attr);
attr.mq_flags |= O_NONBLOCK; /*设置

O_NONBLOCK属性

*/
//attr.mq_flags &= (~O_NONBLOCK); /*取消

O_NONBLOCK属性

*/
mq_setattr(mqd, &attr, NULL)
```

11.2.3 消息的发送和接收

1. 发送消息

POSIX消息队列发送消息和接收消息的接口都很容易理解，从易用性的角度来讲，它们要优于System V消息队列的对应接口。

发送消息的接口定义如下：

```
#include <mqueue.h>
int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned msg_prio);
```

第三个参数`msg_len`表示消息体的长度，长度为0也是合法的，最大不得超过`mq_msgsize`。如果消息体太大，则会返回失败，并置`errno`为`EMSGSIZE`。

第四个参数为消息的优先级，是一个非负的整数。那么问题就来了，容许优先级最大为多少？在Linux中，这个上限为32768。

```
#define MQ_PRIO_MAX    32768
```

如果消息队列已满，`mq_send`函数可能会阻塞。如果设置了`O_NONBLOCK`标志位，这种情况下`mq_send`函数会返回失败，`errno`被置为`EAGAIN`。

2. 接收消息

接收消息的接口定义如下：

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                  size_t msg_len, unsigned *msg_prio);
```

对于POSIX消息队列而言，总是取走优先级最高的消息中最先到达的那个。

第二个参数`msg_ptr`指针用于存放消息体的内存缓冲区的地址，第三个参数`msg_len`是该内存缓冲区的大小。因为消息体的长度是不确定的，所以该缓冲区的大小不得小于最大消息体的长度（`mq_msgsize`），否则一旦消息体长度超过缓冲区的大小，就会失败，并返回`EMSGSIZE`错误。如何获得消息队列的最大消息长度？通过`mq_getattr`函数！

如果第四个参数`msg_prio`不是NULL，那么系统就将取到的消息体的优先级复制到`msg_prio`指向的整型变量。第四个参数如果为NULL，则表示压根不在乎消息体的优先级。

如果调用mq_receive函数时，消息队列中并没有消息，则函数陷入阻塞。如果设置了O_NONBLOCK标志位，则立即返回失败，并设置errno为EAGAIN。

POSIX消息队列的本质就是个优先级队列。优先级高的消息总是被优先取出。从这个角度看，System V消息队列更灵活，它可以让各个进程选取自己感兴趣的消息。

11.2.4 消息的通知

对于System V消息队列，当消息队列里面有消息到来时，消息队列却无法通知其他进程来取。对于消息队列中消息的消费者而言，只有两条路径：

- 调用msgrev函数，阻塞于此，直到消息队列里面有消息。
- 调用msgrev函数时设置IPC_NOWAIT标志位，周期性轮询。

从编程的角度看，期待有这样一种机制来解决上述困境：空的消息队列一收到消息，就给相应进程发出通知，被通知的进程收到通知后就可以及时地处理消息。这种机制称为异步通知机制。POSIX消息队列就引入了这种机制。

POSIX消息队列提供了两种异步通知的方法可供选择：

- 产生一个信号。
- 创建一个线程来执行一个事先指定的函数。

如果一个进程非常关心POSIX消息队列上出现的消息，那么该进程可以通过调用mq_notify函数来表示密切关注。

```
#include <mqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```

mq_notify函数的含义是调用进程通过该接口注册到消息队列，当空消息队列中出现一条消息时，消息队列就会通知到注册进程，也可以通过该接口注销调用进程曾经的注册。手册中英文描述更加准确：

```
mq_notify() allows the calling process to register or unregister for delivery of
an asynchronous notification when a new message arrives on the empty message
queue referred to by the descriptor mqdes.
```

关于消息通知，有以下几个注意事项：

- 只能有一个进程注册到特定的消息队列。如果一个消息队列上已经有注册进程了，那么后续调用mq_notify来注册的进程会返回EBUSY错误。
- 只有在消息进入空消息队列的情况下，才会向注册进程发送通知。如果注册时，消息队列非空，那么只有当消息队列被清空后，又有一条消息到达时，才会发出通知。

·消息队列向注册进程发出通知后，会删除注册信息。之后任何进程都可以通过调用mq_notify函数来注册到消息队列，并接收通知了。

·只有在当前不存在其他进程因在该队列上调用mq_receive（）而陷入阻塞时，注册进程才会收到消息通知。否则阻塞在mq_receive（）上的进程会“截胡”，读取该信息，而注册进程依然保持注册状态。

·进程可以通过在调用mq_notify函数时传入一个值为NULL的sevp参数来撤销自己在消息队列上的注册信息。

前面讨论了消息通知的基本流程，但是当消息队列满足通知的条件时，又是如何通知到注册进程的？

mq_notify函数的关键在第二个入参上，其结构体包含如下参数，若记不清成员变量，则可以通过man sigevent来查看手册。

```
union sigval{
    int sigval_int;
    void *sigval_ptr;
}
struct sigevent {
int sigev_notify; /*决定采用哪种通知方法，信号还是线程

*/
    int sigev_signo; /*用于信号方式，决定发送哪个信号

*/
    union sigval sigev_value; /*信号方式和线程方式都有其独特含义

*/
    void (*sigev_notify_function)(union sigval);
    void *sigev_notify_attributes;
}
```

结构体sigevent的第一个成员sigev_notify用于选择采用哪种方式来通知注册进程，其有效值有以下三个：

·SIGEV_NONE：当消息到达空的消息队列时，不采取任何通知行动。

·SIGEV_SIGNAL：采用发送信号的方式通知进程。

·SIGEV_THREAD：通过调用segev_notify_function中指定的函数来通知进程，就如同在一个新的线程中启动该函数一样。

下面将分别介绍后两种方法。

1.信号通知

如果采用信号方式（SIGEV_SIGNAL），那么调用mq_notify的进程需要约定好希望收到哪种信号，其实现一般如下所示：

```
struct sigevent sev;
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_signo = SIGUSR1;
if(mq_notify(mqd, &sev) == -1) /*mqd为消息队列描述符

*/
{
    /*error handler*/
}
```

调用mq_notify函数的进程需要考虑该如何处理随时可能到来的信号。

最容易想到的方法就是，在信号处理函数中，调用mq_receive函数，并进一步处理消息。很不幸的是，这种方法行不通。第6章讲信号时提到过，大多数函数都不是异步信号安全的，mq_receive函数也不是异步信号安全函数。更何况，还要在信号处理函数中执行复杂的逻辑，这就如同行驶在暗礁丛生的水域，很容易触礁沉船，这种做法是不明智的。

等待信号来临不外乎有以下三种方法：

·sigsuspend

·sigwait

·signalfd

《Linux/Unix系统编程手册》给出了使用sigsuspend函数来等待信号并处理消息的一个例子，而这里我们来介绍下第二种方法，使用sigwait函数来等待信号的来临并处理消息。

在第6章中提到过，sigwait函数的引入，解决了信号的异步带来的很多问题。可以说这个函数提供了一种同步的方式来等待信号的降临。

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

将要等待的信号放置到set中，sigwait函数调用就会被阻塞，直到set集合中的某个信号处于未决状态，sigwait函数才会返回，信号的值记录在sig指针指向的整型变量中。需要注意的一点是，调用sigwait函数之前，set中的所有信号都要被阻塞，否则结果是不可预知的。

以SIGUSR1为例，我们调用mq_notify函数，使消息降临空队列时，发送信号SIGUSR1，主流程等待SIGUSR1，收到信号时，去消息队列中取出该消息，整个流程如下：

```
mqd_t mqd;
struct mq_attr attr ;
sigset_t newmask ;
struct sigevent sigev;
mqd = mq_open(mq_filename,O_RDONLY|O_NONBLOCK);
mq_getattr(mqd,&attr);
buffer = malloc(attr.mq_msgsize);

/*确保

buffer足够大

*/
sigemptyset(&newmask);;
sigaddset(&newmask,SIGUSR1);
sigprocmask(SIG_BLOCK,&newmask,NULL);/*阻塞等待的信号

*/
sigev.sigev_notify = SIGEV_SIGNAL;
sigev.sigev_signo = SIGUSR1;
mq_notify(mqd,&sigev);
for( ; ; )
{
    sigwait(&newmask,&signo);/*等待

SIGUSR1信号

*/
    if(signo == SIGUSR1)
    {
        mq_notify(mqd,&sigev); /*先重新注册

notify函数

*/
        while( n = mq_receive(mqd,buffer,attr.mq_msgsize,NULL) >= 0)
        {
            /*process the message in buffer*/
        }
        if(errno != EAGAIN)
        {
            /*some error happened*/
        }
    }
}
```

需要注意的是，mq_notify函数注册之后，一旦发出信号完成使命，要想继续使用这种通知机制，需要再次调用mq_notify函数重新注册。

使用sigsuspend函数和sigwait函数虽然都可以等到信号的来临，但是也阻塞了当前进程，这并不是明智的做法。更合理的做法是使用signalfd机制，配合select、pool或epoll等多路复用的接口，实现真正

的事件驱动编程。

2.通过线程处理消息

POSIX消息队列提供的另外一种方法就是创建线程，执行预先约定的函数。

在使用中，需要将sigev.sigev_notify设置成SIGEV_THREAD，同时设置好线程应该执行的函数，即将sigev.sigev_notify_function设置成约定好的函数。如果线程函数需要入参，则可以将任何变量的地址填入sigev.sigev_value.sival_ptr中，到达传递参数的目的。

创建的线程具有默认的属性。如果对于线程有特殊的要求，则可以通过如下方法来设置：

```
pthread_attr_t  thread_attr;
pthread_attr_init(&thread_attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
sigev.sigev_notify_attributes = &thread_attr;
```

整体代码流程如下（示意代码，不完整）：

```
static void notify_function(union sigval sv)
{
    struct mqd_t *mqdp = sv.sival_ptr;

    mq_getattr(*mqdp,&attr);
    buffer = malloc(attr.mq_msgsize);

    /*确保

    buffer足够大

    */
    notify_setup(mqdp); /*再次注册

    */
    while(n = mq_receive(*mqdp,buffer,attr.mq_msgsize,NULL)>=0)
    {
        /*处理

        buffer中的消息体

        */
        }
        if(errno != EAGAIN)
        {
            /*发生错误

        */
    }
```

```
    }
    free(buffer);
}
static void notify_setup(mqd_t* mqdp)
{
    struct sigevent sig_ev ;
    sigev.sigev_notify = SIGEV_THREAD;
    sigev.sigev_notify_function = notify_function;
    sigev.sigev_notify_attributes = NULL;
    sigev.sigev_value.sival_ptr = mqdp;
    mq_notify(*mqdp, *sigev);
}
int main()
{
    mqd = mq_open(mqfilename, O_RDONLY | O_NONBLOCK);
    notify_setup(&mqd);
    for(;;)
    {
        pause();
    }
}
```

和信号通知机制一样，一旦创建线程执行完毕，通知机制就结束了，需要重新调用mq_notify函数来注册。

11.2.5 I/O多路复用监控消息队列

POSIX消息队列的通知功能或许在其他Unix平台上非常有用，但是在Linux平台下用处并不大，因为在Linux平台下有更友好、更强大的方法。

在Linux系统中，消息队列描述符被实现成了文件描述符，因此完全可以使用I/O多路复用系统调用来监控消息队列。这种方法非常自然。

《Unix网络编程卷2：进程间通信》的5.6.6节给出了一个例子，如何使用select来监控POSIX消息队列。由于在某些平台下，消息队列描述符并不是文件描述符，所以不能直接使用select。Stevens大师给出的方法就相当地绕，具体方法如下。

首先使用mq_notify函数来注册，确保当空的消息队列中出现消息时，进程会收到信号SIGUSR1；其次进程打开了一个管道，进程调用select监听管道的读取端；在SIGUSR1的信号处理函数中负责往管道的写入端写入一个字符。这样当消息降临空消息队列时，整个的逻辑流程就如图11-2所示。

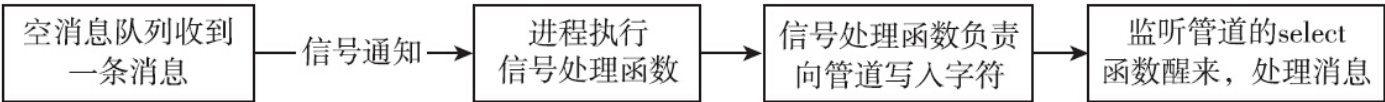


图11-2 APUE中监听消息队列的流程

该方案如此拧巴绝非大师之过，在操作系统不支持的情况下，只能如此处理。因为Linux支持在消息队列上执行select/poll/epoll，所以可以让这条路变得一马平川（如图11-3所示）。

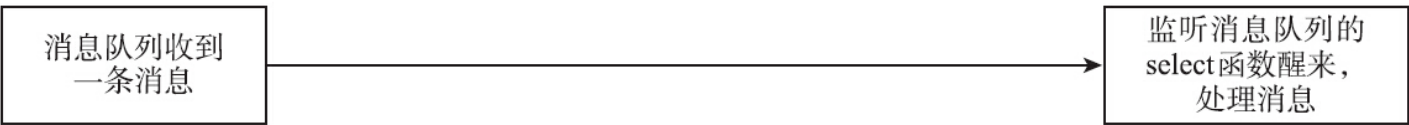


图11-3 Linux下监听消息队列的流程

代码形式如下所示：

```
mqd = mq_open(argv[1],O_RDONLY | O_NONBLOCK);
if(mqd == -1)
{
    fprintf(stderr,"failed to open mqueue (%d: %s)\n",errno,strerror(errno));
    return 1;
}
mq_getattr(mqd,&attr);
buffer = malloc(attr.mq_msgsize);
FD_ZERO(&rset);
for(;;)
{
    FD_SET(mqd,&rset);
    nfds = select(mqd+1,&rset,NULL,NULL,NULL);
    if(FD_ISSET(mqd,&rset))
    {
        while( (n = mq_receive(mqd,buffer,attr.mq_msgsize,NULL)) >=0 )
        {
            /*在此处处理本条消息
        }
    }
    if(errno != EAGAIN)
    {
        /*发生错误，进行错误处理
    }
}
```

注意上面的例子比较简易，仅仅是监听了一个消息队列，根据实际情况，可以同时监听多个消息队列和多个文件。只需要在上面代码的基础上打开其他文件或消息队列，将这些文件描述符置于select的监控之下，如果有来自文件描述符的输入（FD_ISSET来判断），添加相应的处理函数即可。

这条特性并不是标准规定的，标准并未规定将消息队列描述符实现为文件描述符，因此使用I/O多路复用系统调用监控消息队列并不具备可移植性。尽管如此，个人还是认为本条性质是POSIX消息队列最重要的性质，和System V消息队列相比，本条性质给POSIX消息队列带来了压倒性的优势。

11.3 POSIX信号量

POSIX信号量和System V信号量的作用是相同的，都是用于同步进程之间及线程之间的操作，以达到无冲突地访问共享资源的目的。

在前面介绍System V信号量的时候也曾介绍过，Edsger Dijkstra提出了PV操作。所谓P操作，代表荷兰语中的Proberen（意思是尝试），也被称为递减操作或上锁操作。在POSIX术语中为等待（wait）。所谓V操作代表荷兰语单次Verhogen（意思是增加），也被称为递增操作、解锁操作和发信号（signal）操作。在POSIX术语中为挂出（post）。

POSIX信号量的作用和System V信号量是一样的。但是两者在接口上有很大的区别：

- POSIX信号量将创建和初始化合二为一，这就解决了System V中可能出现竞争条件的问题。

- POSIX信号量的修改信号量值的接口（sem_post和sem_wait），一次只能修改一个信号量。与之对应的System V信号量其本质是信号量集，其下的semop函数一次可以修改多个信号量。

- POSIX信号量的修改信号量值的接口（sem_post和sem_wait），一次只能将信号量的值加1或减1。与之对应的System V信号量的semop函数，能够加上或减去一个大于1的值。

- POSIX信号量并没有提供一个等待信号量变为0的接口，而System V信号量中，semop函数则提供了这样的接口。

- POSIX信号量并没有提供UNDO操作，而System V信号量则提供了这样的操作。

从表面看，System V信号量的能力完胜POSIX信号量，事实上并非如此。System V信号量有过度设计之嫌，在大部分场景下，System V提供的第2、3和4条特性都没有什么用处，反而徒增接口的复杂程度。而POSIX信号量提供的接口异常清晰，易于理解和使用。

POSIX信号量真正比System V信号量优越的地方在于，POSIX信号量性能更好。对于System V信号量而言，每次操作信号量，必然会从用户态陷入内核态，可以想象当加锁和解锁操作比较频繁的时候，时间上的开销也是很可观的。POSIX信号量则不然。只要不存在真正的两个线程争夺一把锁的情况，那么修改信号量就只是用户态的操作，并不会牵扯到内核。在竞争并不激烈的情况下，POSIX的性能要远远高于System V信号量。

有得必有失。因为POSIX信号量不会每次操作都去求助内核，所以获得了性能上的提升，但却因此而失去了内核的强大后援。System V信号量支持UNDO操作，当用户进程异常消亡之后，内核会肩负起

为进程还债的责任。但是POSIX信号量却没有这个特性。

POSIX提供了两类信号量：有名信号量和无名信号量。这两种信号量的本质都是一样的，从图11-4可以看出，最重要的sem_wait接口和sem_post接口也都是一样的。如此说来，两种信号量有何不同呢，各自应用在哪些场景呢？

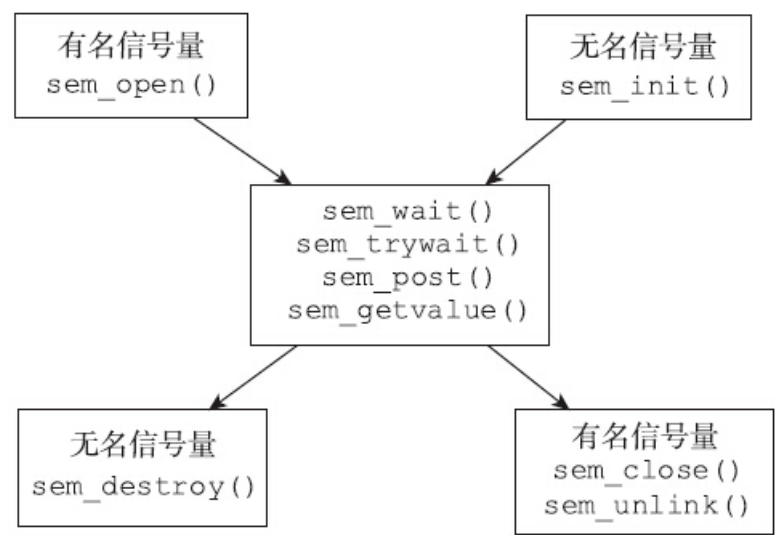


图11-4 有名信号量和无名信号的接口

无名信号量，又称为基于内存的信号量，由于其没有名字，没法通过open操作直接找到对应的信号量，所以很难直接用于没有关联的两个进程之间。无名信号量多用于线程之间的同步。

有名信号量由于其有名字，多个不相干的进程可以通过名字来打开同一个信号量，从而完成同步操作，所以有名信号量的操作要方便一些，适用范围也比无名信号量更广。

下面将分别介绍这些接口。

11.3.1 创建、打开、关闭和删除有名信号量

创建或打开有名信号量，需要调用`sem_open`函数，其接口定义如下：

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

第二个参数`oflag`标志位支持的标志包括`O_CREAT`和`O_EXCL`标志位。如果带了`O_CREAT`标志位，则表示要创建信号量。

`mode`表示创建的新信号量的访问权限，标志位和`open`函数一样，`mode`参数的值也会根据进程的`umask`来取掩码。

`value`是新建信号量的初始值。创建和赋初值都是由一个接口来完成的，这样就不会出现System V信号量可能出现的初始化竞争的问题了。`value`的值在最小值0和最大值`SEM_VALUE_MAX`之间。SUSv3要求最大值至少等于32767，对于Linux而言，这个限制为`INT_MAX`（在Linux/x86平台上，该值是2147483647）。

当`sem_open`函数失败时，返回`SEM_FAILED`，并且设置`errno`。

注意，不要尝试创建`sem_t`结构体的副本，下面这段代码的做法是错误的：

```
sem_t *sem_p, sem_dup;
sem_p = sem_open(...

);
sem_dup = *sem_p; /*非法操作

*/
sem_wait(&sem_dup);
```

上面定义了`sem_p`的副本`sem_dup`，但在副本上执行`sem`的相关操作，行为是不可预知的，不要这样使用。切记，后面所有的调用都要用通过`sem_open`返回的`sem_t`类型的指针来进行操作，而不能使用结构体的副本。

当一个进程打开有名信号量时，系统会记录进程与信号的关联关系。调用`sem_close`时，会终止这种关联关系，同时信号量的进程数的引用计数减1。关闭信号量的接口定义如下：

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

进程终止时，进程打开的有名信号量会自动关闭。当进程执行exec系列函数时，进程打开的有名信号量会自动关闭。

但是关闭不等同于删除，如果要删除信号量则需要调用sem_unlink函数，其接口定义如下：

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

将有名信号量的名字作为参数，传递给sem_unlink，该函数会负责将该有名信号量删除。由于系统为信号量维护了引用计数，所以只有当打开信号量的所有进程都关闭了之后，才会真正地删除。

11.3.2 信号量的使用

信号量的使用，总是和某种可用资源联系在一起的。创建信号量时的value值，其实指定了对应资源的初始个数。当申请该资源时，需要先调用sem_wait函数；当发布该资源或使用完毕释放该资源时，则调用sem_post函数。

1.等待信号量

sem_wait函数用于等待信号量，它会将信号量的值减1，其接口定义如下：

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

如果调用sem_wait函数时，信号量的当前值大于0，那么sem_wait函数立刻返回。否则sem_wait函数陷入阻塞，待信号量的值大于0之后，再执行减1操作，然后成功返回。

如果陷入阻塞的sem_wait函数被信号中断，则返回-1，并且置errno为EINTR。使用sigaction注册信号处理函数时，无论是否使用了SA_RESTART标志位，都不会自动重启系统调用。

如果仅仅是尝试等待信号量，而不想陷入阻塞，则可以调用sem_trywait函数，其接口定义如下：

```
int sem_trywait(sem_t *sem);
```

sem_trywait会尝试将信号量的值减1，如果信号量的值大于0，那么该函数将信号量的值减1之后会立刻返回。如果信号量的当前值为0，那么sem_trywait也不会陷入阻塞，而是立刻返回失败，并置errno为EAGAIN。

若资源当前不可得，那么sem_wait调用就可能会陷入无限期阻塞，而sem_trywait调用则选择立刻返回失败，绝不阻塞。除了这两种选择，系统还提供了第三种选择：有限期等待，即sem_timedwait函数。

sem_timedwait函数的接口定义如下：

```
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

第二个参数为一个绝对时间。可以使用gettimeofday函数获取到struct timeval类型的当前时间，然后将timeval转换成timespec类型的结构体，最后在该值上加上想等待的时间。或者调用clock_gettime函数，直接获得timespec结构体类型的变量表示当前时刻，然后在结构体上加上想等待的时间，作为第二个参数传给sem_timedwait函数。

如果超过了等待时间，信号量的值仍然为0，那么返回-1，并置errno为ETIMEOUT。

2.发布信号量

sem_post函数用于发布信号量，表示资源已经使用完毕，可以归还资源了。该函数会使信号量的值加1。

sem_post接口定义如下：

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

如果发布信号量之前，信号量的值是0，并且已经有进程或线程正等待在信号量上，此时会有一个进程被唤醒，被唤醒的进程会继续sem_wait函数的减1操作。如果有多个进程正等待在信号量上，那么将无法确认哪个进程会被唤醒。

当函数调用成功时，返回0；失败时，返回-1，并置errno。当参数sem并不指向合法的信号量时，置errno为EINVAL；当信号量的值超过上限（即超过INT_MAX）时，置errno为EOverflow。

3.获取信号量的值

sem_getvalue函数会返回当前信号量的值，并将值写入sval指向的变量，代码如下：

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

如果信号量的值大于0，含义自不必说；但是如果信号量的值等于0，同时又有很多进程或线程阻塞在信号上，那么应该返回0还是返回一个负值——其绝对值等于等待进程的个数？看起来后者更有意义，因为从该值可以获知到竞争的激烈程度，但是Linux还是选择返回0。

当sem_getvalue返回时，其返回的值可能已经过时了。从这个意义上讲，该接口的意义并不大。

11.3.3 无名信号量的创建和销毁

无名信号量，由于其没有名字，所以适用范围要小于有名信号量。只有将无名信号量放在多个进程或线程都共同可见的内存区域时才有意义，否则协作的进程无法操作信号量，达不到同步或互斥的目的。所以一般而言，无名信号量多用于线程之间。因为线程会共享地址空间，所以访问共同的无名信号量是很容易办到的事情。或者将信号量创建在共享内存内，多个进程通过操作共享内存的信号量达到同步或互斥的目的。

1. 初始化无名信号量

无名信号量的初始化是通过`sem_init`函数来完成的。

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

其中，第二个`pshared`参数用于声明信号量是在线程间共享还是在进程间共享。`0`表示在线程间共享，非零值则表示信号量将在进程间共享。要想在进程间共享，信号量必须位于共享内存区域内。

无名信号量的生命周期是有限的，对于线程间共享的信号量，线程组退出了，无名信号量也就不复存在了。对于进程间共享的信号量，信号量的持久性与所在的共享内存的持久性一样。

无名信号量初始化以后，就可以像操作有名信号量一样操作无名信号量了。

2. 销毁无名信号量

销毁无名信号量的接口定义如下所示：

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

`sem_destroy`用于销毁`sem_init`函数初始化的无名信号量。只有在所有进程都不会再等待一个信号量时，它才能被安全销毁。对Linux实现而言，省略`sem_destroy`函数，也不会带来异常。但是为了安全性和可移植性，还是应该在合适的时机正常销毁信号量。

11.3.4 信号量与futex

使用POSIX信号量，链接的时候需要加上-lpthread，而不是-lrt。由此可以看出POSIX信号量与NPTL线程库渊源甚深。

第7章讲线程时曾提到过，互斥量是建立在快速用户空间互斥体（英文全名为fast userspace mutex，简称futex）基础上的。POSIX信号量也是架构在futex基础之上的。

快速用户空间互斥体，是一种用户态和内核态协同工作的同步机制。同步的进程需要一段共享内存，futex变量就位于这段内存之中。当进程尝试进入或退出互斥区时，首先会检查共享内存中的futex变量，如果没有竞争发生，则原子地修改futex变量，无须执行系统调用。如果通过访问futex变量的值发现有竞争发生，则执行相应的系统调用来完成相应的处理。

对于线程间同步，因为同一个进程下的多个线程共享该进程的地址空间，所以同时操作某个futex变量并不是特别难以做到的事情。如果是用于进程间的同步，则首先需要一块内存空间，而且要让多个进程都可以操作该内存空间，这就牵扯到共享内存了。事实上调用sem_open函数来创建POSIX信号量时，使用了后面会介绍到的mmap，并在多个进程之间共享文件的内容。

下面的代码摘自glibc的sem_open函数：

```
/* Create the initial file content.  */
union
{
    sem_t initsem;
    struct new_sem newsem;
} sem;
/*信号量的初始值为

value. 后面会写入文件

*/

sem.newsem.value = value;
sem.newsem.private = 0;
sem.newsem.nwaiters = 0;
memset ((char *) &sem.initsem + sizeof (struct new_sem), '\0',
        sizeof (sem_t) - sizeof (struct new_sem));
...
/*将

sem相关的值写入文件，并通过

mmap映射到进程的内存中

*/

if (TEMP_FAILURE_RETRY (__libc_write (fd, &sem.initsem, sizeof (sem_t)))
    == sizeof (sem_t))
    /* Map the sem_t structure from the file.  */
    && (result = (sem_t *) mmap (NULL, sizeof (sem_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0)) != MAP_FAILED)
```

每创建一个名为name的信号量，在/dev/shm下就会多出一个名为sem.name的文件。该文件的内容是sem_t结构体：

```
#if __WORDSIZE == 64
# define __SIZEOF_SEM_T 32
#else
# define __SIZEOF_SEM_T 16
#endif
typedef union
{
    char __size[__SIZEOF_SEM_T];
    long int __align;
} sem_t;
```

在x86架构下，32位系统里，该结构体的大小是16字节，在x86_64架构下，该结构体的大小是32字节。事实上，真实存放的内容是new_sem结构体：

```
union
{
    sem_t initsem;
    struct new_sem newsem;
} sem;
/*new_sem是真正使用的结构体

*/
struct new_sem
{
    unsigned int value; /*当前信号量的值
```

```
*/
int private;
unsigned long int nwaiters;
};
```

下面创建一个名为res_88的信号量，创建该信号量时，将信号量的值初始化为88。代码如下所示：

```
sem_t* sem = sem_open(argv[1], O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR, 88);
```

我们可以通过查看/dev/shm/sem.res_88来查看该信号量的情况：

```
manu@manu-rush:~$ od -x /dev/shm/sem.res_88
0000000 0058 0000

0000 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000040
```

其文件内容的含义如图11-5所示。



图11-5 /dev/shm/sem.name文件的内容

从输出中的00580000（0x58等于88）可知，当前信号量的值是88。

当将信号量的值减少到零，并且有两个进程在等待信号量时：

```
manu@manu-rush:~$ od -x /dev/shm/sem.res_88
0000000 0000 0000 0000 0000 0002 0000 0000 0000 0000

0000020 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

输出中的0002 0000 0000 0000（0x02）表示当前有两个进程等待在该信号量上。

对于POSIX信号量而言，需要同步的进程通过mmap将文件内容映射进了进程的地址空间。对这段内存的修改，其他进程也可见。

内核提供了futex系统调用，其接口定义如下：

```
#include <linux/futex.h>
#include <sys/time.h>
int futex(int *uaddr, int op, int val, const struct timespec *timeout,
          int *uaddr2, int val3);
```

第一个参数uaddr是用户空间的一个地址，里面存放的是整型变量。

第二个参数op用于存放操作命令，最基本的两个操作命令FUTEX_WAIT和FUTEX_WAKE。

当op是FUTEX_WAIT时，会原子地检查uaddr地址存放的int值是否等于val，如果是，那么内核会使进程陷入休眠，同时把进程挂到uaddr对应的等待队列上。

当op是FUTEX_WAKE时，最多唤醒val个等待在uaddr上的进程。

在没有竞争的情况下，可以通过实验来比较下System V信号量和POSIX信号量的效率，见表11-6。同样是初始化一个信号量的值为1，然后交替执行wait和post操作各100万次，可以看出，System V信号量花费的时间是POSIX信号量的40多倍。

表11-6 POSIX信号量和System V信号量在无竞争情况下的性能比较

	POSIX 信号量	System V 信号量	System V 信号量 (UNDO)
real	0m0.109s	0m4.793s	0m4.934s
user	0m0.104s	0m2.472s	0m2.548s
sys	0m0.005s	0m2.303s	0m2.381s

用strace统计系统调用次数，可以看到System V信号量共执行了200万次semop系统调用，而POSIX信号量只有两次futex，事实上，这仅有的两次futex系统调用，也与sem_post和sem_wait调用无关。

% time	seconds	usecs/call	calls	errors	syscall
0.86	0.000012	6	2	1	futex
% time	seconds	usecs/call	calls	errors	syscall
99.99	14.265818	7	2000000		semop

网上有些文章认为sem_post无论是否存在竞争都会执行futex系统调用，很明显这种观点是错误的，通过简单的实验不难验证这点。因为信号量的数据结构中记录了等待者的数量，因此不难判断是否需要执行系统调用，来唤醒等待者。

至于存在竞争的情况，在互斥量相关的章节中已经介绍过，这里就不再赘述。

11.4 内存映射mmap

消息队列和信号量都已经介绍过了，按照正常的逻辑，本节应该介绍POSIX共享内存，为什么这里却要介绍内存映射mmap呢？

这是因为内存映射mmap是POSIX共享内存的基础，内存映射完成了大量的基础性工作，临门一脚交给了共享内存。事实上POSIX共享内存也要和mmap配合使用。不理解mmap就不能很好地理解POSIX共享内存。

更重要的是，纵然不提共享内存，mmap这个系统调用也是非常重要的，其重要程度远远超过POSIX共享内存。只要你在Linux平台上工作，每天就一定会执行无数次的mmap系统调用，不管是直接地还是间接地。

当你执行哪怕是最简单的ls命令时，mmap系统调用在背后都会默默地帮你加载动态链接库，当你调用malloc函数分配大于MMAP_THRESHOLD大小（默认是128KB）的内存时，mmap系统调用会躲在malloc背后支撑；当你调用pthread_create创建线程时，mmap系统调用会帮你分配好线程栈；当你创建POSIX信号量时，mmap会默默帮你开辟一段空间存放futex变量.....

可能迄今为止你从未在代码中直接使用mmap，但它就静静地躺在那里，对你的帮助不增也不减。

11.4.1 内存映射概述

mmap系统调用的作用是在调用进程的虚拟地址空间中创建一个新的内存映射。根据内存背后有无实体文件与之关联，映射可以分成以下两种：

- 文件映射：内存映射区域有实体文件与之关联。mmap系统调用将普通文件的一部分内容直接映射到调用进程的虚拟地址空间。一旦完成映射，就可以通过在相应的内存区域中操作字节来访问文件内容。这种映射也被称为基于文件的映射。
- 匿名映射：匿名映射没有对应的文件。这种映射的内存区域会被初始化成0。

一个进程映射的内存可以与其他进程中的映射共享物理内存。所谓共享是指各个进程的页表条目指向RAM中的相同分页。如图11-6所示。

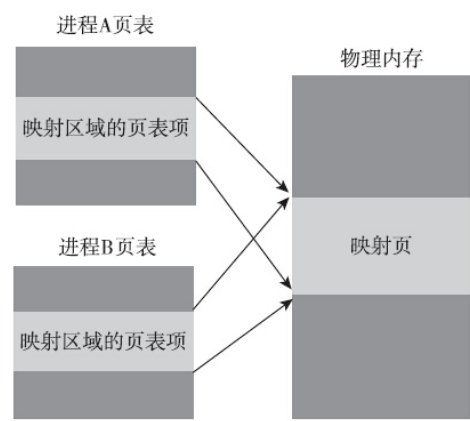


图11-6 进程内存共享映射

这种内存映射的共享，会在以下两种情况下发生：

- 通过fork，子进程继承了父进程通过mmap映射的副本。
- 多个进程通过mmap映射了同一个文件的同一个区域。

无论映射背后有无实体文件与之关联，这个进程之间共享映射的特性都是非常有用的。我们知道，进程的虚拟地址空间是彼此隔离的，一个进程不能直接操作另一个进程虚拟地址空间中的内存。但是mmap系统调用给出了两个办法，让多个进程可以共享一片内存区域。

看到第一种方式，即通过fork子进程继承父进程通过mmap映射的副本，大家的心中可能会隐隐有种不安。第4章曾提到过，虽然子进程拷贝了父进程的内存，但是父子进程的页表并不是始终都指向同一物理内存的，一旦父子进程中有一个尝试修改内存的内容时，内核就不得不发起写时复制，分配新的物理内存。从此父子进程分道扬镳，彼此再也看不到对方对内存的改动。

对于进程malloc出来的内存，栈上的变量的确如此，fork之后父子进程并不是共享同一块映射。但是通过mmap系统调用创建的内存映射却可以做到进程之间共享同一个内存映射。当然进程之间要不要共享映射也是可以选择的，这取决于该映射是私有映射还是共享映射。

- 私有映射（MAP_PRIVATE）：在映射内容上发生的变更对其他进程不可见。对于文件映射而言，变更不会同步到底层文件中。对映射内容所做的变更是进程私有的。事实上，内核使用了写时复制技术来完成这个任务。未对映射内容进行修改操作时，页面仍然是共享的。一旦有进程试图修改其中一个分页的内容时，内核首先会为该进程创建一个新的分页，并将需要修改的分页中的内容拷贝到新分页中。
- 共享映射（MAP_SHARED）：在映射内容上发生的所有变更，对所有共享同一个映射的其他进程都可见。对于文件映射而言，变更会同步到底层的文件中。很明显，共享映射是用于进程间通信的。

内存映射根据有无文件关联，分成文件与匿名；根据映射是否在进程间共享，分成私有和共享。这两个维度两两组合，内存映射共分成4种类型，其各自的用途如表11-7所示。

表11-7 内存映射的分类及用途

变更可见范围	映 射 类 型	
	文 件	匿 名
共享	内存映射 IO，进程间共享内存	进程间共享内存
私有	根据文件内容初始化内存	内存分配

下面，我们开始介绍如何利用mmap接口，实现这四种不同的内存映射。

11.4.2 内存映射的相关接口

mmap函数的接口定义如下：

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

这个函数的参数比较多。其中fd、offset和length这三个参数指定了内存映射的源，即将fd对应的文件，从offset位置起，将长度为length的内容映射到进程的地址空间。

对于文件映射，调用mmap之前需要调用open取到对应文件的文件描述符。

第一个参数addr用于指定将文件对应的内容映射到进程地址空间的起始地址。一般来讲为了可移植性，该参数总是指定为NULL，表示交给内核去选择合适的位置。

第三个参数prot用于设置对内存映射区域的保护，它的合法值及其含义如表11-8所示。

表11-8 mmap调用中prot的合法值及其含义

prot	说 明
PROT_EXEC	映射的内容可以执行
PROT_READ	映射的内容可以读取
PROT_WRITE	映射的内容可以修改
PROT_NONE	映射的内容不可访问

flags参数用于指定内存映射是共享映射还是私有映射，也用于指定内存映射是文件映射还是匿名映射。flags可选的标志位及含义如表11-9所示。

表11-9 mmap调用中flags可选的标志位及含义

标 志 位	说 明
MAP_SHARED	请求创建共享映射
MAP_PRIVATE	请求创建私有映射
MAP_ANONYMOUS	请求创建匿名映射，fd 参数必须是 -1

其中调用mmap函数时，MAP_SHARED和MAP_PRIVATE标志位，两者必须指定一个。

flags中另一个可选的标志位是MAP_FIXED。如果指定了该标志位，那么表示函数调用者铁了心要把内容映射到对应的地址上。这种情况下，addr一般要求按页对齐。如果内核无法映射文件到该指定位置，则调用失败。如果地址和长度指定的内存区域和已有映射有重叠部分，那么重叠区的原始内容将被丢弃，然后填入新的内容。使用该选项需要非常了解进程的地址空间，否则不建议使用。

需要注意的是mmap系统调用的操作单元是页。参数addr和offset都必须按页对齐，即必须是页面大小的整数倍。在Linux下，页面大小是4096字节，该值可以通过getconf命令来获取到：

```
getconf PAGESIZE
4096
```

对于编程接口，Linux提供了sysconf函数来获取到相关配置项的值：

```
#include <unistd.h>
long sysconf(int name);
```

对于获取页面大小而言，可以通过如下代码获取到页面的大小：

```
long pagesize = sysconf(_SC_PAGESIZE);
```

在进程的地址空间里，映射区域总是页面的整数倍。但是有些时候，`mmap`传递的`length`值并非页面的整数倍，比如文件映射时，文件的大小或要映射进内存的区域并非页面的整数倍，这时候，`mmap`会按照页面的大小向上取整，多出来的内存区域（最后一个有效字节到映射区域边界）会填充0。

当`mmap`调用成功时，则返回映射区域的起始地址，如果失败，则返回`MAP_FAILED`，并置`errno`。

如果不再需要对应的内存映射了，可以调用`munmap`函数，解除该内存映射：

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

其中`addr`是`mmap`返回的内存映射的起始地址，`length`是内存映射区域的大小。执行过`munmap`后，如果继续访问内存映射范围内的地址，那么进程会收到`SIGSEGV`信号，引发段错误。需要注意的是，关闭对应文件的文件描述符并不会引发`munmap`。

如果创建内存映射时`flags`中带上了`MAP_PRIVATE`标志位，那么解除该内存映射时，调用进程对内存映射的所有改动都会被丢弃。

介绍完基本接口，下面将分别介绍4种不同的映射，以及它们的应用场景。

11.4.3 共享文件映射

1.共享文件映射的建立和使用

创建共享文件映射的步骤如下所示。

- 1) 打开文件，获取文件描述符`fd`，这一步是通过`open`来完成的。
- 2) 将文件描述符作为`fd`参数，传给`mmap`函数。

整个步骤如下面的伪代码所示：

```
fd = open(...);  
addr = mmap(..., MAP_SHARED, fd, ...);  
close(fd);    /*可选，可以关闭，也可以不关闭
```

```
*/
```

第1)步打开文件时设置的权限必须要和`mmap`系统调用需要的权限相匹配。具体来讲就是：

·打开时，必须允许读取，即`O_RDONLY`和`O_RDWR`至少要指定一个。

·`mmap`调用时，如果`prot`参数中指定了`PROT_WRITE`，并且`flags`中指定了`MAP_SHARED`，那么打开时，必须带有`O_RDWR`标志位。

`open`时需要注意，并非所有的文件都支持`mmap`操作，比如管道文件就不支持`mmap`操作。

`mmap`完成之后关闭文件描述符并不会导致内存映射被解除，因此，在没有其他需要的情况下，可以调用`close`关闭文件。

但在某些场景下，后续操作还会用到文件描述符，因此不宜关闭文件。比如进程需要将对内存映射所做的修改立刻同步到底层的磁盘文件中，这可能需要对文件描述符`fd`调用`fsync`或`fdatasync`；再比如多个进程间共享内存映射，它们都要修改内存映射的部分内容，这种情况可以通过文件的记录锁（`fcntl`函数的`F_SETLK`命令）来同步进程的操作。因此，建立共享文件映射之后是否关闭文件描述符，需要根据实际情况来做决定。

`mmap`这个接口容易产生一个误解是，调用`mmap`时，真的已经把文件对应区域的内容读取到了内存的对应位置。事实上并非如此，`mmap`仅仅是建立了两者之间的关联。当第一次读取映射区的内容或修改映射区的内容时，会引发缺页中断（`page fault`），这时候才会真正地将文件的内容加载到内存的对应位置。

当`mmap`调用成功之后，共享映射在进程地址空间中的位置，以及和对应文件的关系如图11-7所示。

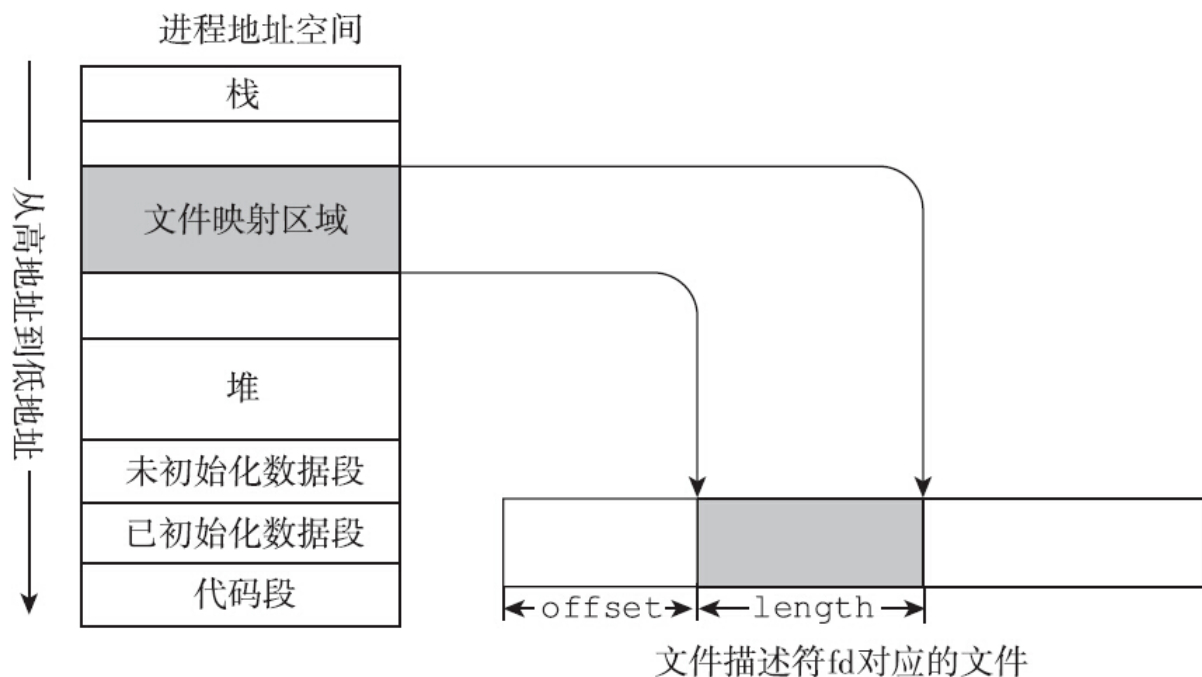


图11-7 进程地址空间中映射区域和文件的关系

文件是有长度的，所以正常情况下offset和length参数应该遵循一定的限制：offset应小于文件的长度，并且offset+length也应小于文件的长度。很有意思的是，mmap函数并不检查offset和size定义的区域是否在文件的范围之内。示例代码如下：

```
#define MB (1024*1024)

ret = fstat(fd,&stat_buf);
if(ret < 0 )
{
}
off_t filesize = stat_buf.st_size ;
off_t offset = (filesize % PAGE_SIZE == 0) ? \
    filesize : (filesize/PAGE_SIZE + 1)*PAGE_SIZE;
mmap_base = mmap(NULL,stat_buf.st_size+MB,PROT_READ,MAP_SHARED,fd,offset);
if(mmap_base == MAP_FAILED)
{
    fprintf(stderr,"failed to mmap (%s)\n",strerror(errno));
    ret = 3;
    goto out ;
}
```

上面的代码中，将文件结尾之后的1M字节映射到进程的地址空间，映射的区域和文件完全没有交集。在这种情况下，mmap也不会因offset和length参数而返回MAP_FAILED，而是正常地返回。



注意 此处说的是不检查offset和length定义的范围是否在文件长度范围之内，并不是说不检查offset和size的值。mmap调用要求offset必须为系统分页的整数倍，这个限制始终存在。如若offset的值不是系统分页的整数倍，mmap会返回MAP_FAILED，并置errno为EINVAL。

尽管mmap不检查对应区域是否落在文件的长度范围之内，但是这并不意味着随意建立的映射也能正常使用。使用共享文件映射需要谨慎，否则很容易触发错误。

最容易想到的一种错误就是没有映射某区域却强行访问，而且无论该区域是否落在文件的长度范围以内（如图11-8所示）。这种访问会引发段错误，产生SIGSEGV信号。该信号的默认动作是进程终止并产生核心转储文件。

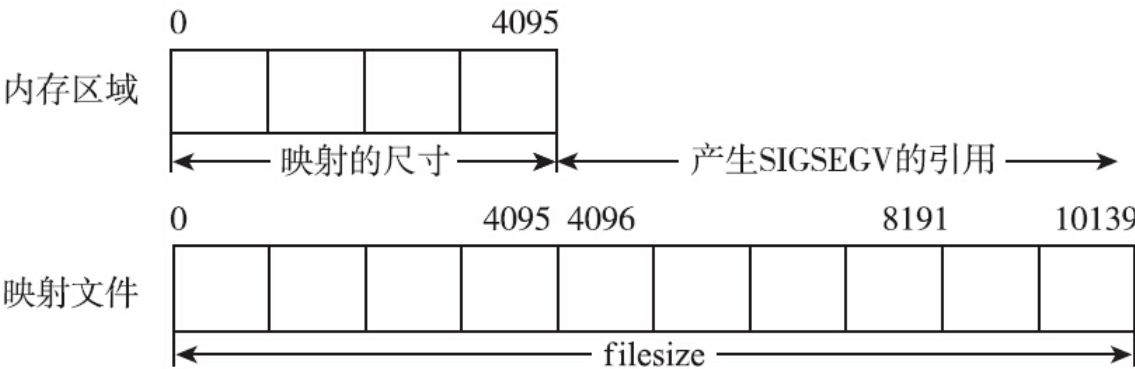


图11-8 访问内存映射的常见错误（1）

这种错误是一目了然的。但是如果调用mmap时length不是系统分页大小（4KB）的整数倍时，情况就会稍稍有些复杂，如图11-9所示。文件的长度为10KB，但是调用mmap时，将文件的前5KB映射到了进程的地址空间。这种情况下，真正映射的大小会被向上舍入成系统分页的整数倍，对于这个例子而言，虽然mmap调用指定了5KB，但是真实映射了8KB的大小。用户访问mmap返回基地址偏移8KB之内的内存地址，都不会触发SIGSEGV信号。访问基地址偏移8KB之后的地址，才会触发SIGSEGV信号。

另外一种错误是访问的映射地址虽然在mmap映射的内存区域之内，但并不在文件长度的范围以内（如图11-10所示），这种情况会导致SIGBUS信号的产生，该信号的默认动作也是进程终止并产生核心转储文件。这种错误之所以会出现时因为mmap并不会检查offset和size定义的区域是否落在文件长度范围以内。既然建立映射的时候不检查，那么真正访问对应内存地址的时候，就可能触发错误。

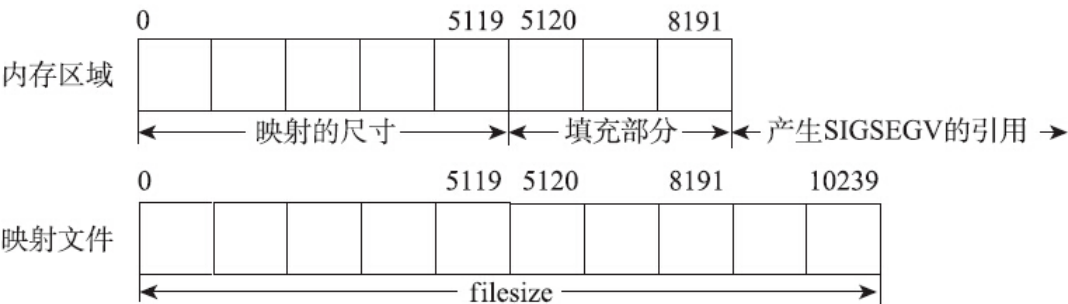


图11-9 访问内存映射的常见错误（2）

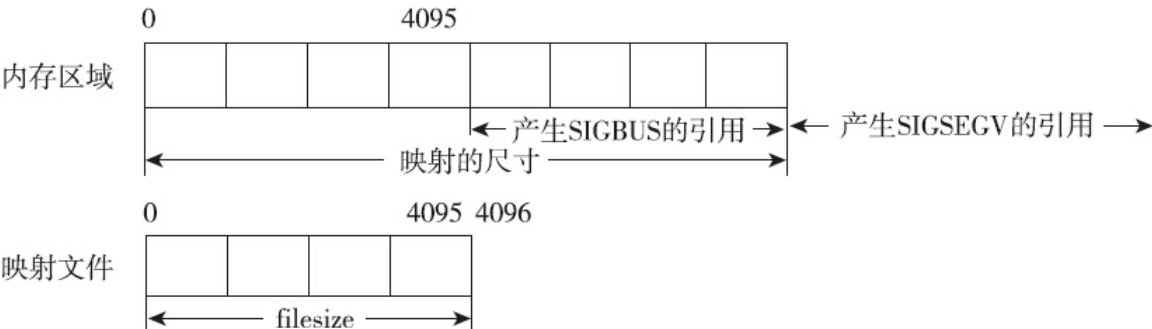


图11-10 访问内存映射的常见错误（3）

这种错误也是很明显的。但是当文件的大小不是系统分页整数倍时，也会带来一定的特殊情况，如图11-11所示。

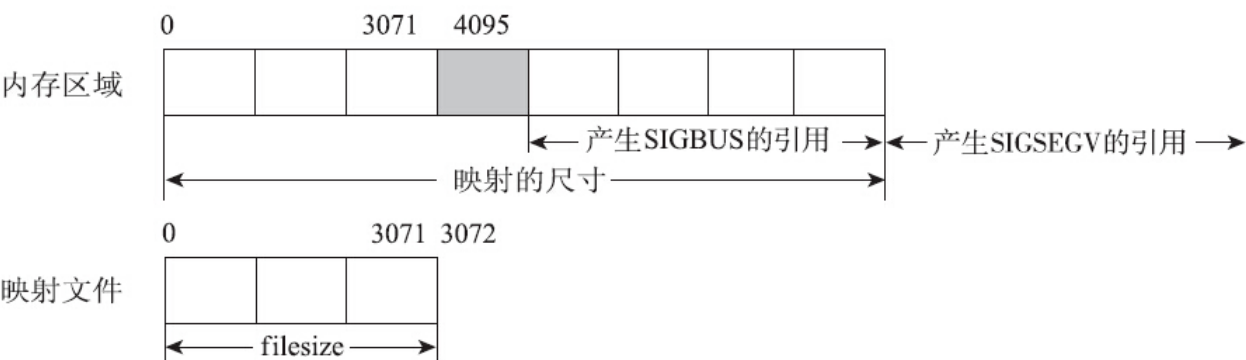


图11-11 访问内存映射的常见错误（4）

尽管文件的长度是3KB，但是mmap映射了一个长度为8KB的内存区域。4KB~8KB这个范围自不必说，超出了文件的范围，访问时一定会触发SIGBUS信号。但是比较挠头的是3KB~4KB这个范围的内存。因为这个范围已经不在文件的长度范围之内了，却又和文件的有效映射同处一个页面。这种情况下允许访问，而且不会触发SIGBUS信号。至于要访问8KB之后的内存，那已经是尝试访问映射范围之外的内存了，会触发上一种错误，即产生SIGSEGV信号。

第一种情况即访问映射范围之外的内存属于典型的作死小能手的行为。但是很有意思是，有时候访问本映射范围之外的内存却不一定会触发SIGSEGV。原因是mmap区域可能存在多个内存映射，虽然超出了本想访问的映射的范围，却歪打正着访问到了相邻的内存映射。注意，这种情况并不值得窃喜，而是更糟糕，因为程序已经不是按照预设的逻辑在运行了，继续运行很可能会在某个不可预知的地方崩溃，这就增大了排查问题的难度。

第二种错误更需要程序员小心。表面看只要调用mmap时小心谨慎，不主动出幺蛾子（即映射超出文件长度范围的内存区域），SIGBUS信号就不会出现。其实则不然。如果内存映射在使用过程中，调用truncate或ftruncate将文件截断，那么访问文件真实长度之外的区域，就会触发SIGBUS信号（如图11-12所示）。因为共享文件映射始终和文件相关联，因此这种情况要小心防范。

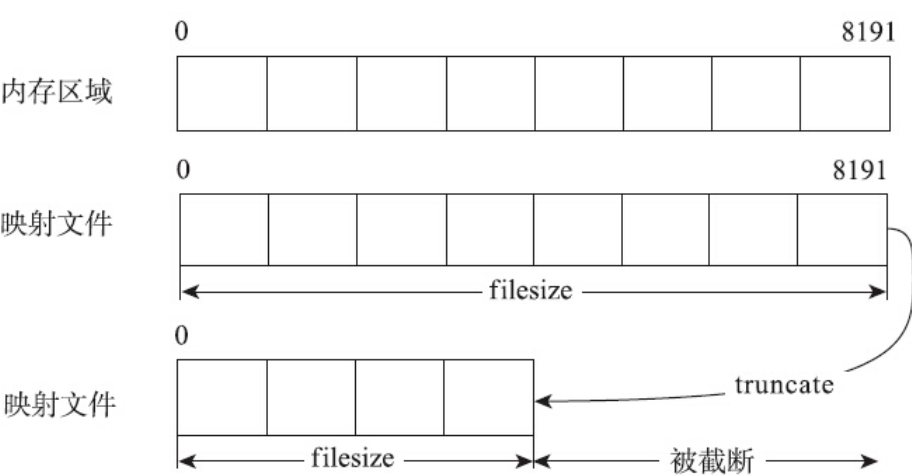


图11-12 truncate文件引起SIGBUS信号的产生

2.共享文件映射的用途

共享文件映射主要用于两个方面：操作文件和进程间通信。

Linux提供了read、write、lseek等操作文件的系统调用，通过这些接口可以操作文件。共享文件映射给出了另外一种操作文件的方法。

共享文件映射将文件的内容映射到了进程的地址空间。对应区域中的内容来源于文件，对映射内容所做的修改，都会自动反应到文件上，内核会负责将修改最终同步到底层的块设备。因此共享文件映射区域的内存，就等同于对文件的读写。

访问过的文件页面，很可能还会继续访问。不同进程很可能会访问同一文件页面。如果每次访问文件的内容，都要操作底层块设备，那性能就会很差。因此现代的操作系统都提供了文件缓存，Linux也不例外。Linux提供了页高速缓存（Page Cache，也称页缓存）用以减少对磁盘的访问。

在大部分情况下，应用程序都会通过页高速缓存来读写文件。当读取文件的某一部分内容时，内核首先会从页高速缓存中查找所读取的数据是否存在对应的页面，如果请求的页面不在页高速缓存之中，那么内核就会负责分配页面并添加到页高速缓存中，然后从磁盘上读取对应的数据来填充它。如果物理内存足够大，空闲页面足够多，那么该页将长期保留在页高速缓存中，使得其他进程访问该页数据时不需要再访问磁盘。当应用程序向文件写入时，会直接修改页高速缓存中的数据，但是并不会立刻写入磁盘，而是将该页标记成脏页，由内核负责在合适的时机将脏页回写到磁盘中。

调用read也好，调用write也罢，事先都要准备用户空间缓冲区buffer，如图11-13所示。读取时，将读到的内容复制到该buffer中；写入时，再将buffer中的内容写入文件中。

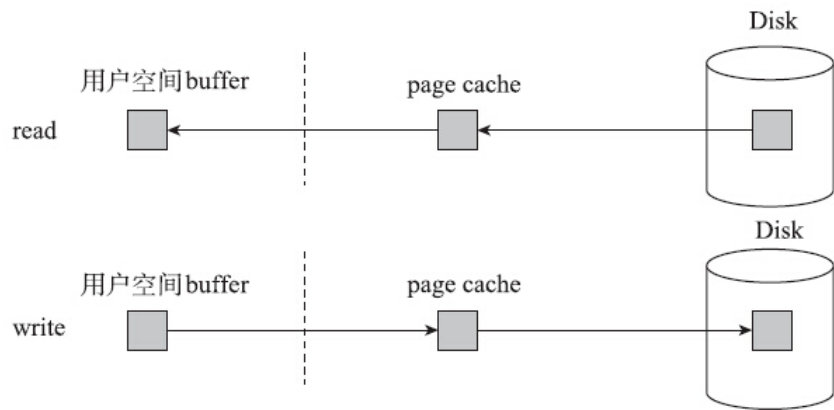


图11-13 read和write都需要用户空间缓冲区

对于read和write接口而言，姑且不论磁盘与页高速缓存之间如何交互，页高速缓存和用户空间缓冲区之间的数据传输是不可避免的。但是如果使用mmap来操作文件，则不需要这次复制。mmap对共享文件映射的操作，直接作用在页高速缓存上，节省了一次数据传输。

这是不是意味使用mmap来操作文件要比使用read/write的性能更好呢？大家很容易产生这种想法，但这种想法有些想当然。随着硬件的发展，内存拷贝消耗的时间已经极大地降低了，可是mmap访问文件内容，会引起缺页中断（page fault）。相对于内存拷贝而言，缺页中断的开销更大，加上创建内存映射、解除内存映射及更新硬件内存管理单元的翻译后备缓冲器（TLB）的开销，大部分情况下（不考虑刻意构造的场景），mmap的性能反而要低于read和

write。

共享文件映射的第二个用途是进程间通信。进程的地址空间是彼此隔离的，一个进程一般不能直接访问另一个进程的地址空间。通过共享文件映射，两个进程的映射区域指向了同一个物理内存（即前面提到的页高速缓存），这就给进程间通信提供了可能，如图11-14所示。如果两个进程的共享文件映射都源自同一个文件的同一个区域，那么一个进程对映射区域的修改，对于另外一个进程是立刻可见的，同时内核会负责在合适的时机将修改同步到底层文件。之所以能够做到这点，是因为两个映射区域的对应分页都指向了同一个页高速缓存（Page Cache），下一小节将会介绍内核的相关实现。

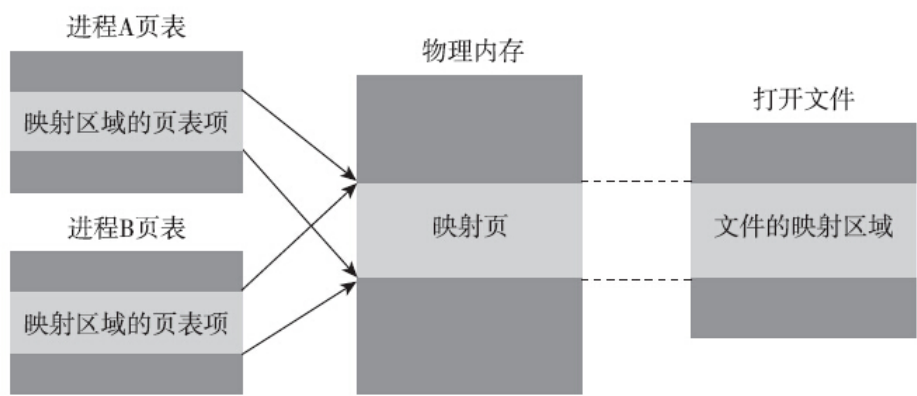


图11-14 进程通过共享文件映射共享物理内存

所有的共享内存都会遇到的问题是同步。无论是System V信号量还是POSIX信号量，都可以用于同步对共享内存的操作。除此以外，记录锁也比较适用于操作共享文件映射。

fcntl函数提供了记录锁的功能，和flock函数提供的文件锁功能相比，fcntl提供的记录锁可以提供更细粒度的控制。flock函数提供的锁是粗放型锁，锁定的是整个文件，无法锁定文件的某个区域。fcntl提供的锁可以锁定文件的某个区域，如图11-15所示，这样就减少了因竞争而陷入阻塞的概率，从而提高了性能。

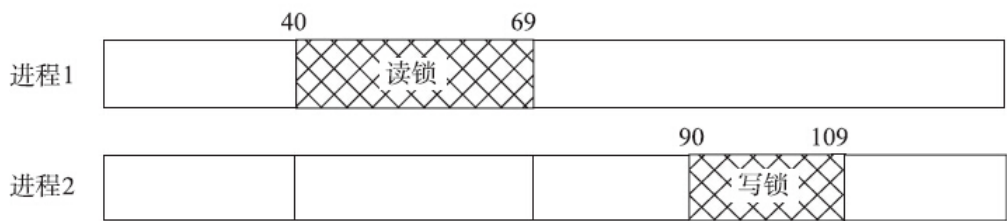


图11-15 记录锁可对文件的特定区域上锁

fcntl函数接口的定义如下所示：

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

其中与记录锁相关的cmd为：

- F_SETLKW：尝试锁定文件的对应区域。如果该区域已经被锁定，则陷入阻塞。
- F_SETLK：尝试锁定文件的对应区域。如果该区域已经被锁定，则立刻返回-1。

·F_GETLK: 仅仅是查询锁的信息，并不会真正地对某区域加锁。

当执行加锁相关操作时，需要用到flock结构体，代码如下：

```
struct flock {
    ...    short l_type;    short l_whence;    off_t l_start;    off_t l_len;    pid_t l_pid;    ...
}
```

其中l_type用于指定锁的类型，以及指定解锁操作，其合法值及其含义如下：

·F_RDLCK: 读锁

·F_WRLCK: 写锁

·F_UNLCK: 解锁

l_whence的含义和lseek函数的第三个参数whence的含义一样，表示如何解释偏移量，有效值有SEEK_SET、SEEK_CUR和SEEK_END。l_whence参数结合l_start和l_len参数，定义了文件的某个区域。

使用fcntl对文件的某个区域加锁解锁的方法如下示例代码所示：

```
struct flock fl;
int ret;
fl.l_type = F_WRLCK;
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = 100;
/*对文件对应区域加锁

*/
ret = fcntl(fd, F_SETLK, &fl);
/*访问或修改

[0, 99] 范围内的文件内容

*/
/*对文件对应区域解锁

*/
fl.l_type = F_UNLCK;
ret = fcntl(fd, F_SETLK, &fl);
```

通过上面的讨论可以看出，fcntl提供的记录锁非常适用于同步共享文件映射的操作。可以轻易地做到读写请求分开，以及更细粒度、更灵活的控制。



注意 flock和fcntl都属于劝告式锁（Advisory Lock），如果同步的进程遵循游戏规则，操作之前先申请锁，就能起到同步的作用；但是如果进程无视劝告式锁的存在，不遵循游戏规则，不申请锁直接操作文件或文件的某个区域，内核也不会阻止这种操作。

3.共享文件映射的内核实现

对于共享文件映射而言，最大的谜团是：进程地址空间彼此独立，互不干扰，可是多个进程通过mmap映射同一

文件的同一区域时，却指向了同一物理页面，修改彼此可见。内核是如何做到的？

前面已经提到过，答案是通过页高速缓存。在追踪mmap内核实现之前，首先来简单介绍下页高速缓存。

引入页高速缓存的目的是为了性能。现在访问的文件的某个页面，将来可能还会再访问。如果不将页面缓存进内存，那么每次读取文件，就都不得不操作慢速的块设备，这会极大地影响性能。

页高速缓存该如何组织多个页面，以便在需要时可以快速定位到这些缓存页面呢？对于单个文件来说，有些文件系统支持TB级别的文件（比如ext4文件系统就已经支持16TB的单个文件了），4KB一个页面的情况下，页面的数目是巨大的。如果不能高效地组织页面，那么花费在查找页面上的时间就可能会很长，届时纵然页面已经在缓存中，也会因查找缓存页面太慢而导致性能的急剧恶化。内核使用了基数树（radix tree）来解决这个难题，如图11-16所示。

只要找到文件对应的基数树的根，就可以快速定位到与文件对应的页面（如果它在页高速缓存中的话）上。现在问题就转变成了：当进程操作文件时，如何快速找到与文件对应的基数树。

对于这个话题，毛德操老爷子在《Linux内核情景分析》一书的5.6节中有高屋建瓴的分析。内核文件层有三个核心的数据结构：file、dentry和inode。虽然三种数据结构都可以通过各种指针来跳转，找到与文件对应的页高速缓存，但是inode是和页高速缓存关系最密切的数据结构。struct file数据结构是进程层面的概念，提供的是目标文件的一个上下文信息。对于同一个文件，不同进程可以在该文件上建立不同的上下文，甚至同一个进程也可能因多次打开文件而建立起多个上下文。换句话说，数据结构struct file和实体文件并不是一对一的关系，而是多对一的关系。dentry结构体虽不是进程层面的概念，但是dentry和实体文件也不是一对一的关系，通过文件连接，可以为已存在的文件建立别名。只有inode结构最适合和文件的页缓存关联，因为inode和实体文件是一对一的关系。

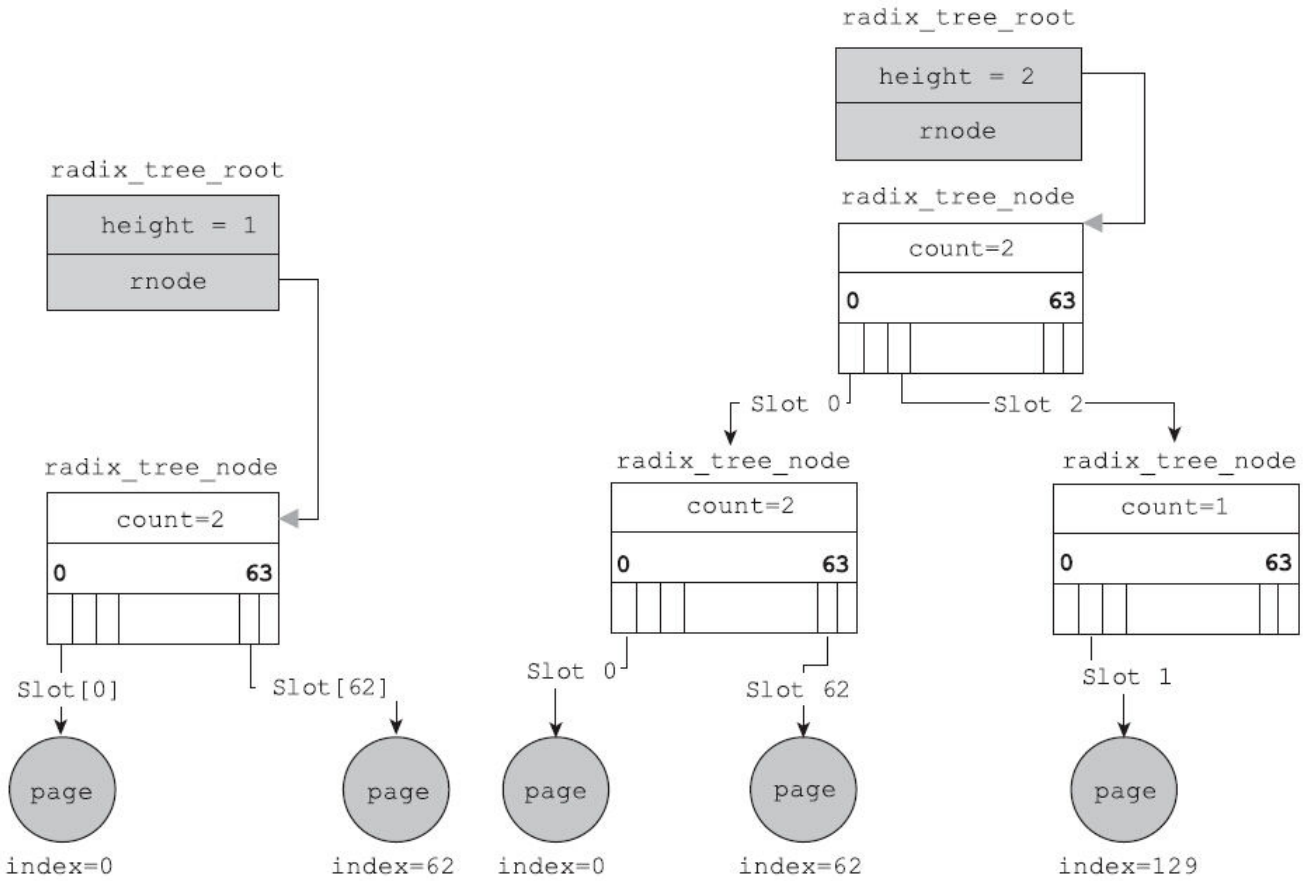


图11-16 通过基数树加速页面查找

图11-17给出了内核中文件与页缓存相关的数据结构。从图11-17不难看出，当进程通过文件系统接口（read/write等），不难找到与文件对应的inode。Linux内核引入了地址空间address_space这个数据结构来管理页高速缓存。inode中的i_mmaping成员变量指向对应的address_space结构体。不论多少个进程通过文件系统API来操作文件，也不论多少个进程通过mmap建立共享文件映射来操作文件，同一个文件只对应一个address_space结构体。通过该数据结构就能找到与页高速缓存对应的基数树，进而找到对应的缓存页（如果存在的话）。

通过图11-17可以很清晰地看出，当通过文件系统接口进行读写时，如何找到与文件对应的缓存页面。但是mmap内存映射区域和页高速缓存如何建立联系却并不明晰。下面我们跟踪mmap系统调用的实现来一探究竟。

调用mmap函数，进入内核之后首先会执行到arch/x86/kernel/sys_x86_64.c中的如下函数：

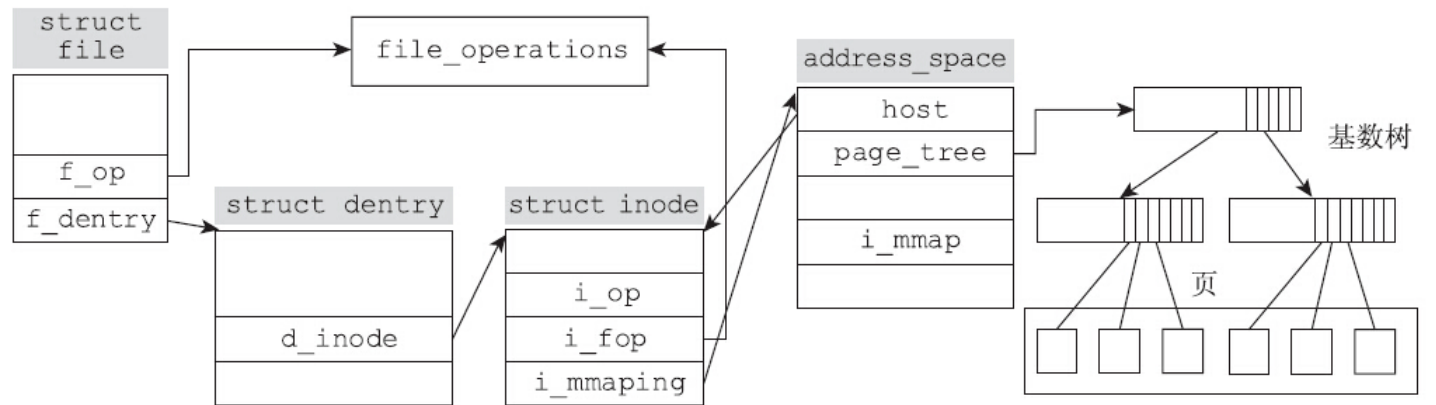


图11-17 文件与页缓存相关的数据结构

```
SYSCALL_DEFINE6(mmap, unsigned long, addr, unsigned long, len,  
                unsigned long, prot, unsigned long, flags,  
                unsigned long, fd, unsigned long, off)
```

该函数非常简单，把绝大部分工作都委托给了内核的sys_mmap_pgoff函数。该函数定义在mm/mmap.c中，其原型声明如下：

```
SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr, unsigned long, len,  
                unsigned long, prot, unsigned long, flags,  
                unsigned long, fd, unsigned long, pgoff)
```

这个函数是分析mmap实现的起点。而该函数将大部分工作都委托给了do_mmap_pgoff。该函数的总体流程如图11-18所示。

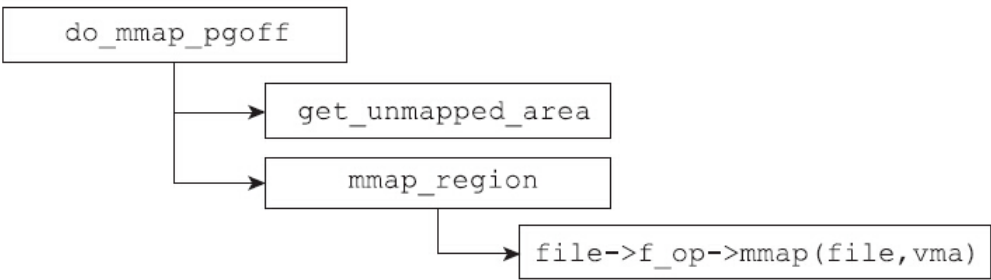


图11-18 内核do_mmap_pgoff调用关系

内核为了管理进程的地址空间，引入了虚拟内存区域（virtual memory area，VMA）的数据结构。

vm_area_struct结构体描述了进程地址空间内一个独立的内存范围，如图11-19所示。当通过mmap函数创建一个共享文件映射（当然不仅仅是共享文件映射）的时候，内核就会为进程分配一个新的vm_area_struct结构体。每一个vm_area_struct都对应进程地址空间中的唯一一个内存区间。其中成员变量vm_start指向区间的开始地址（vm_start本身属于对应内存区间），vm_end指向内存区间的结束地址（vm_end本身不属于对应内存区间），vm_end减去vm_start的值即为内存区间的长度。对于共享文件映射而言，该长度为调用mmap时指定的length向上取整为页面大小的整数倍。

vm_area_struct结构体中的vm_flags成员变量记录的是该内存区域的VMA标志位。该标志位记录了对应内存区域的一些属性。比如VM_READ标志位表示对应的页面可读取；VM_WRITE标志位表示对应的页面可写；VM_EXEC标志位表示对应的页面可执行；VM_LOCKED表示对应的页面被锁定，等等。

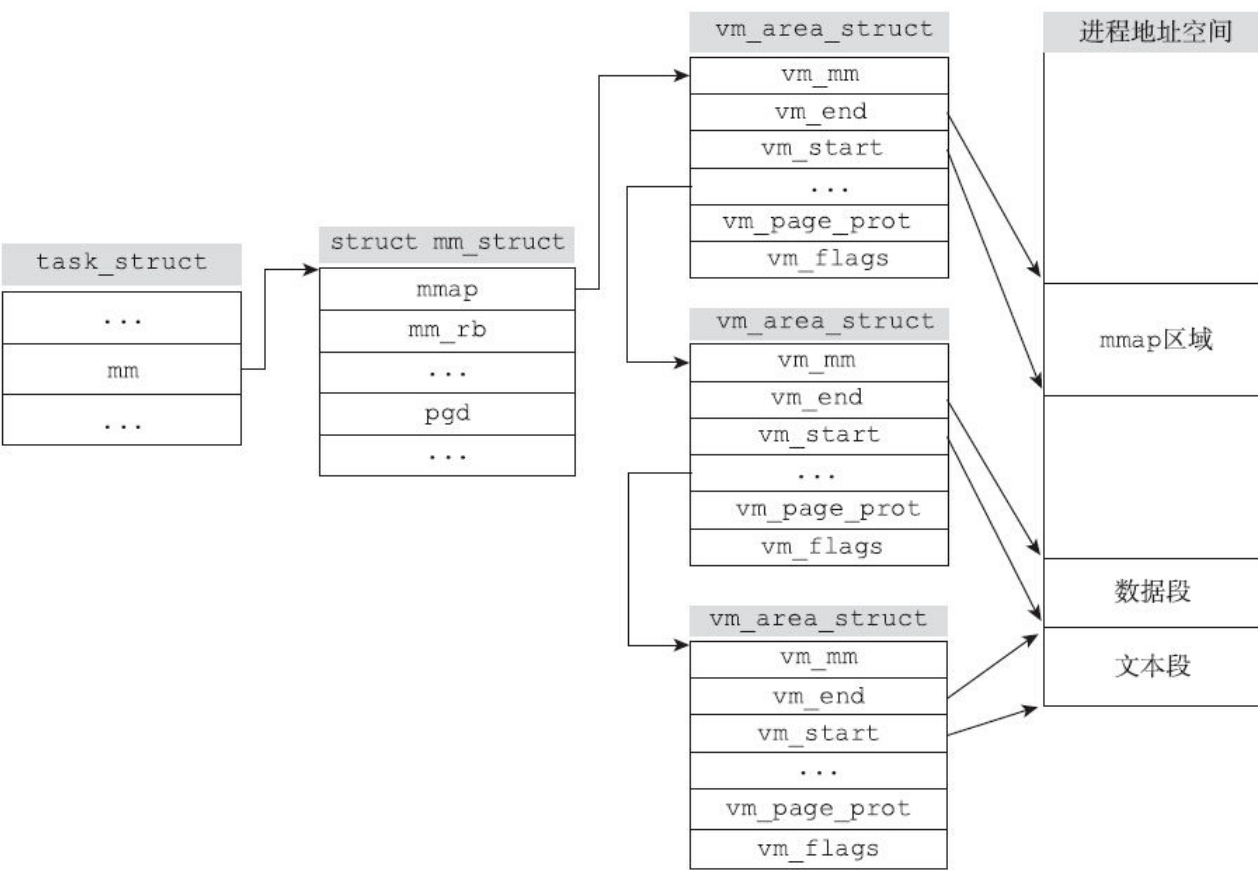


图11-19 进程地址空间与VMA

如果虚拟内存区域和文件相关联，那么vm_area_struct结构体中的vm_file成员变量就指向与文件对应的struct file结构。通过该指针，虚拟内存区域就可以和文件发生关联。

另外一个很重要的成员变量是vm_ops。该成员是一个指针，指向与内存区域相关的操作函数。

```
struct vm_operations_struct {
    ...
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    ...
}
```

因为vm_area_struct是一个通用的数据结构，可以代表任意类型的内存区域，因此不同的VMA就有不同的操作函数，vm_ops也就指向了不同的操作函数集合。

VMA操作函数集合中的fault函数用于应对这种场景：访问的页面并没有出现在物理内存中；而page_mkwrite用于

应对页面为只读，应用程序却尝试写入的情况。这两个函数都会被缺页中断处理程序调用，以处理不同的情景。

下面以主流的ext4文件系统为例，追踪一下整个流程。ext4文件系统中inode的i_fop注册成ext4_file_operations。

ext4_file_operation的定义位于fs/ext4/file.c中，其中与mmap相关的操作函数定义如下：

```
const struct file_operations ext4_file_operations = {
    .mmap      = ext4_file_mmap,
}
```

当mmap系统调用在mmap_region中执行file->f_op->mmap（file, vma）时，执行的就是ext4_file_mmap函数。因此对于ext4文件系统而言，文件映射的调用路径就变成了如图11-20所示的路径。

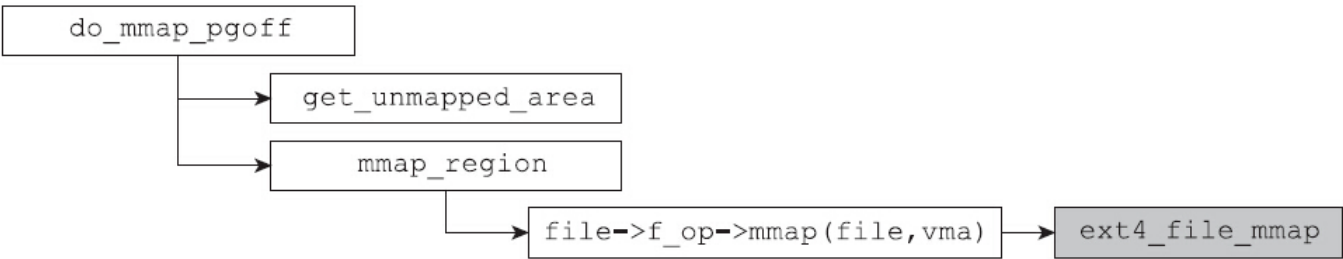


图11-20 内核do_mmap_pgoff的调用关系（2）

该函数异常简单，简单到我不介意将全函数都贴在这里：

```
static int ext4_file_mmap(struct file *file, struct vm_area_struct *vma)
{
    struct address_space *mapping = file->f_mapping;
    if (!mapping->a_ops->readpage)
        return -ENOEXEC;
    file_accessed(file);
    vma->vm_ops = &ext4_file_vm_ops;
    vma->vm_flags |= VM_CAN_NONLINEAR;
    return 0;
}
```

这个ext4_file_map函数仅仅安装了一个内存区操作函数，即把vma的vm_ops指针指向了ext4_file_vm_ops。

```
static const struct vm_operations_struct ext4_file_vm_ops = {
    .fault      = filemap_fault,
    .page_mkwrite = ext4_page_mkwrite,
};
```

注意整个mmap调用的过程中并没有对文件的大小做过判断。换言之，哪怕文件的大小只有100个字节，mmap仍然可以将文件映射到1MB的内存空间。下面的示例代码中，尽管映射了比文件的大小还要多1MB的空间，但是mmap调用依然会成功。

```
/*省略了

error handler*/
#define MB_1 (1024*1024)
fd = open(argv[1], O_RDONLY);
ret = fstat(fd, &stat_buf);
mmap_base = mmap(NULL, stat_buf.st_size+MB_1, PROT_READ, MAP_SHARED, fd, 0);
if (mmap_base == MAP_FAILED)
{
}
```

mmap之后，尽管进程的虚拟地址空间内已经有一块内存区域和文件相对应，但内核并没有将文件的内容加载到

内存区域。将虚拟内存区域vma的vm_ops指向ext4_file_vm_ops实例，其实是埋下了伏笔。一旦将来需要访问映射区域的页面，尽管物理内存中没有，但依然可以依靠VMA操作函数集里的对应函数来处理这个危机。

知乎上有一个提问是“有哪些老鸟程序员知道而新手不知道的小技巧”，该提问下有一个很意思很有良心的回复：

把觉得不靠谱的需求放到最后做。很可能到时候需求就变了。

——知乎用户mu mu

这个技巧在计算机科学上也被广泛地使用着。写时复制采用的是这种思想，接下来要介绍的请求调页也是如此。在操作系统领域，未雨绸缪从来不是一个褒义词，因为这往往意味着会做大量的无用功。

请求调页是一种动态内存分配技术，该技术把页面的分配推迟到不能再推迟为止。也就是说，一直推迟到进程要访问的地址不在物理内存为止。这项技术的核心思想是，进程开始运行时并不会访问其地址空间的所有地址，事实上，有些地址进程可能永远都不会访问。

一旦用户访问映射的内存区域，就会触发缺页中断。以arch/x86/mm/fault.c中的do_page_fault为起点，其调用路径如图11-21所示。

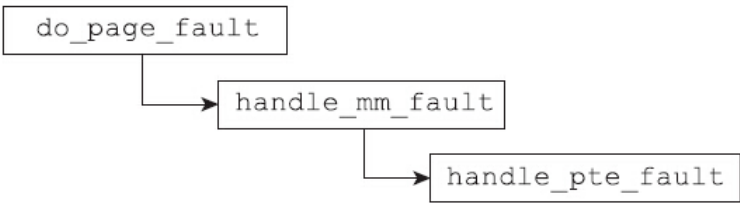


图11-21 缺页中断的函数调用关系

在handle_pte_fault中有如下代码：

```
entry = *pte;
if (!pte_present(entry)) {
    /*pte_none表示没有对应页表项，内核需要从头开始加载该页

*/
    if (pte_none(entry)) {
        /*若是基于文件的映射，则请求调页

*/
        if (vma->vm_ops) {
            if (likely(vma->vm_ops->fault))
                return do_linear_fault(mm, vma, address,
                    pte, pmd, flags, entry);
        }
        /*若是匿名映射，则按需分配

*/
        return do_anonymous_page(mm, vma, address,
            pte, pmd, flags);
    }
    /*如果该页标记不存在，但是页表中保存了相关的信息，则表示该页已被换出

*/

    /*非线性映射换出部分，不能像普通页那样换入，必须恢复非线性关联

*/
```



```

if (pte_file(entry))
    return do_nonlinear_fault(mm, vma, address,
        pte, pmd, flags, entry);
return do_swap_page(mm, vma, address,
    pte, pmd, flags, entry);
}

```

`pte_present(entry)` 用于判断页面是否在物理内存中。如果页面并不在物理内存中，那么处理流程如图11-22所示。

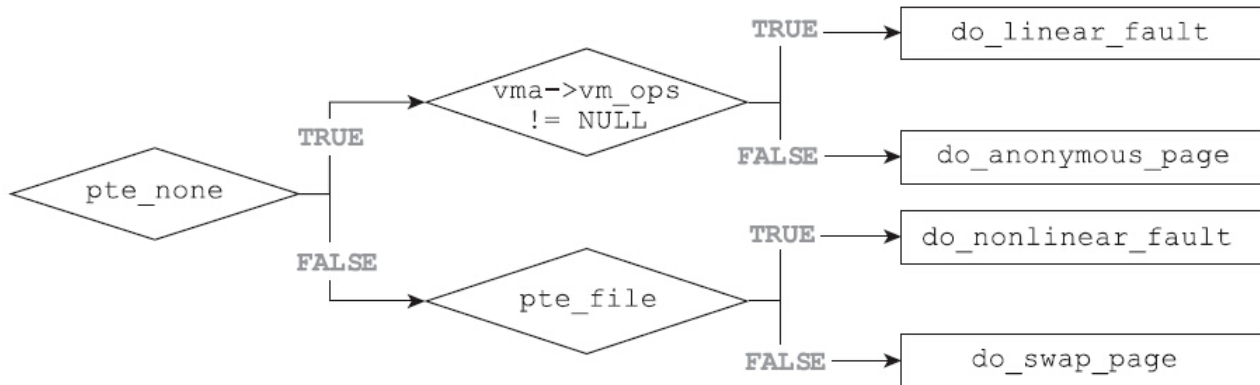


图11-22 页面不在物理内存时的处理流程

`pte_none`用于判断是否存在对应的页表项。如果`pte_none`为true，那么内核必须从头开始加载该页。这种情况下，根据vma的`vm_ops`是否注册了`vm_operation_struct`而分成两类：如果`vm_ops`不是NULL，则表示是基于文件的映射，就会调用`do_linear_fault`；如果`vm_ops`等于NULL，则表示是匿名映射，内核就会调用`do_anonymous_page`来返回一个匿名页。

如果`pte_none`返回false，则表示页表中保存了相关的信息，这就意味着该页已经被换出，这种情况下应该调用`do_swap_page`从系统的某个交换区换入该页。

但是有一种特殊情况，即`pte_file`函数返回true的情况。`pte_file`函数用于检测页表项是否属于非线性映射，如果该函数返回true，则表示页表项属于非线性映射。所谓非线性映射是指在mmap的基础上分离的映射页。尽管映射的内容仍然是文件的内容，但是与映射区域对应的并不是文件的连续区间，实际情况是每一个内存页都映射的是文件数据的随机页。对于应用程序而言，要想建立非线性映射，首先需要调用`mmap`创建常规的、连续的内存映射，然后调用`remap_file_pages`来重新映射某些页面。对于非线性映射而言，已经换出的部分不能像普通页一样被换入，首先必须正确地恢复非线性关联。这种特殊情况，是由`do_nonlinear_fault`函数负责处理的。

对于共享文件映射，调用的是`do_linear_fault`函数。在该函数中会执行`vma->vm_ops->fault`函数。如果对应的文件属于ext4文件系统，那么mmap系统调用中已经将vma的`vm_ops`指定成了`ext4_file_vm_ops`，因此，`vma->vm_ops->fault`指向的就是`filemap_fault`函数。

`filemap_fault`函数是非常重要的，不仅仅是ext4文件系统，还有很多文件系统都使用`filemap_fault`来处理缺页。该函数不仅可以读入所需的数据，还实现了预读的功能。接下来，我们来分析下`filemap_fault`函数。

```

int filemap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
    int error;
    struct file *file = vma->vm_file;
    struct address_space *mapping = file->f_mapping;
    struct file_ra_state *ra = &file->f_ra;
    struct inode *inode = mapping->host;
    pgoff_t offset = vmf->pgoff;

```

```

struct page *page;
pgoff_t size;
int ret = 0;
size = (i_size_read(inode) + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
if (offset >= size)
    return VM_FAULT_SIGBUS;
page = find_get_page(mapping, offset);

```

当多个进程mmap同一文件的某个区域时，当操作映射区域时，更多的情况是该页面已经在页缓存之中了。因此filemap_fault首先会调用file_get_page来检查请求页面是否已经在页缓存之中了。

如果页缓存中确实不存在请求的页面，则需要调用page_cache_read将内容从底层块设备中读取上来，其函数定义如下：

```

static int page_cache_read(struct file *file, pgoff_t offset)
{
    struct address_space *mapping = file->f_mapping;
    struct page *page;
    int ret;
    do {
        page = page_cache_alloc_cold(mapping);
        if (!page)
            return -ENOMEM;
        ret = add_to_page_cache_lru(page, mapping, offset, GFP_KERNEL);
        if (ret == 0)
            ret = mapping->a_ops->readpage(file, page);
        else if (ret == -EEXIST)
            ret = 0; /* losing race to add is OK */
        page_cache_release(page);
    } while (ret == AOP_TRUNCATED_PAGE);
    return ret;
}

```

mapping->a_ops是什么？创建ext4 inode的ext4_create函数中有如下一句代码：

```
ext4_set_aops(inode);
```

在这个函数中我们通过上述语句设置了mapping的aops。对于readpage函数而言，最终是通过ext4_readpage调用了通用函数mpage_readpage。如图11-23所示。

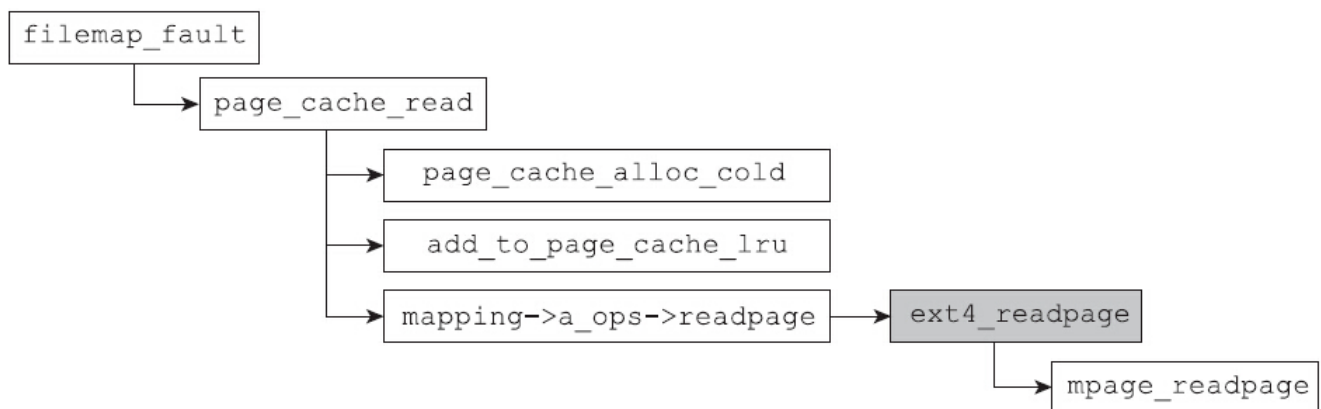


图11-23 缺页中断当page cache中不存在请求页面时的调用路径

正是因为页缓存的存在，才真正做到了当多个进程mmap同一个文件的某个区域时，其指向的物理内存是同一个分页。

事实上，系统文件的所有共享都是基于同一条路线的，不论你是read、write还是mmap，都要遵循这条路线：系统唯一的文件路径到系统唯一的inode，再到相同的address_space，最后到相同的页面。

我们跟踪了文件映射的内核实现，得到的结论是页缓存是联系内存管理系统和文件系统的一条纽带。应用层无论

是使用read/write系统调用还是mmap将文件映射到内存，都是基于页缓存的，殊途同归。因此通过映射获取的文件视图和通过I/O系统调用（read、write）获得文件视图是一致的。

理解了这个问题，我们就可以讨论如下这类的话题了：如果mmap引入的共享文件映射，修改了映射区的内存后，进程却意外死亡，那么进程对内存的修改能否同步到底层文件？答案是肯定的，页高速缓存到底层文件的冲刷（flush）是由内核来负责的。事实上，我们不难验证这一点。对这个话题感兴趣的话，可以阅读[stackoverflow](http://stackoverflow.com/questions/5902629/mmap-msync-and-linux-process-termination)上的相关文章 [1]。

关于共享文件映射，另外一个很有意思的现象是：修改映射区的内存，哪怕是几个字节，也可能需要花费很长的时间（比如几百毫秒） [2]。很多人都遇到了这个问题 [3]，原因是内核回写线程会负责将脏页回写，它会将正在回写的页设置成写保护。此时如果有用户进程对该页面执行写操作，就会因为碰到了写保护的页面而走到do_page_fault。这种情况下，最终会执行到handle_pte_fault中的如下语句：

```
if (flags & FAULT_FLAG_WRITE) {
    if (!pte_write(entry))
        return do_wp_page(mm, vma, address,
                           pte, pmd, ptl, entry);
    entry = pte_mkdirty(entry);
}
```

在do_wp_page函数中会调用page_mkwrite方法，在这里会等待回写线程写完之后才可以完成对页面的写操作。参考文献已经介绍得非常详细了，限于篇幅此处就不展开了。

[1] <http://stackoverflow.com/questions/5902629/mmap-msync-and-linux-process-termination>。

[2] 写mmap内存变慢的原因：<http://oldblog.donghao.org/2012/02/mmapauaeaayao000.html>。

[3] mmap internals：<http://blog.chinaunix.net/uid-20662820-id-3873318.html>。

11.4.4 私有文件映射

当调用mmap时，如果将flags设置成MAP_PRIVATE标志位，那么映射就是私有文件映射。最常见的情况就是前面提到的加载动态共享库，多个进程共享相同的文本段。

从下面执行ls时执行的系统调用中可以看出:

```
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\30\2\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1807032, ...}) = 0
mmap(NULL, 3921080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fe89a1a1000
```

一般来讲文本段通常被保护成PROT_READ|PROT_EXEC。为了防止恶意程序篡改内存上的保护信息之后再篡改程序或共享库的文本，通常会直接使用私有文件映射而不是共享文件映射。

相对于出现得更早的静态库，动态库有很多的优点：可执行文件变得更小，节省磁盘空间；内存中只需要一份共享库的实例，不同进程都可以使用因而节省了内存。

11.4.5 共享匿名映射

和文件映射相对应的是匿名映射。这种映射并没有文件与之对应。一般来讲创建匿名映射有两种方法：

- 调用mmap时，在参数flags中指定MAP_ANONYMOUS标志位，并且将参数fd指定为-1。
- 打开/dev/zero设备文件，并将得到的文件描述符fd传递给mmap。

不论采用哪种方式，得到的内存映射中的字节都会被初始化成0。

本节介绍共享匿名映射，下一节将介绍私有匿名映射。调用mmap创建匿名映射时，如果flags设置了MAP_SHARED标志位，那么创建出来就是共享匿名映射。共享匿名映射的作用是让相关进程共享一块内存区域。

比如父进程创建一个共享匿名映射，然后fork创建子进程，这种情况下，父子进程就可以通过这块内存区域来通信。这个过程的代码如下所示。

```
addr = mmap(NULL, length, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, -1, 0);
if (addr == MAP_FAILED)
{
    /*error handle*/
}
child_pid = fork()
```

11.4.6 私有匿名映射

当创建匿名映射时，如果flags中设置了MAP_PRIVATE标志位，那么创建出来的内存映射就是私有匿名映射。

这种映射最典型的用途是分配进程所需的内存。映射出来的内存并没有文件与之关联，对内存的操作也是私有的，不会影响到其他进程。该用途比较典型的例子就是glibc中的malloc实现。当要分配的内存大于MMAP_THRESHOLD字节时，glibc的malloc是使用mmap来实现的。一般来讲该阈值是128KB，可以通过mallopt函数来调整该参数。

当代码中有如下内容时：

```
char *p = malloc(128*1024);
```

通过strace来跟踪程序的执行，我们可以清楚地看到程序调用了mmap系统调用：

```
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2a29f0c000
```

对于malloc的glibc实现感兴趣的话，Justin N.Ferguson的《Understanding the heap by break it》是一篇非常好的参考文献。

11.5 POSIX共享内存

前面曾经讲述过，`mmap`系统调用做了大量的工作，POSIX共享内存和前面的共享文件映射相比，并没有什么特殊之处。如果非要说有差别，那么差别就是，获取文件描述符的方式不同。

普通文件映射获取

fd的方式

```
fd = open(filename, ...);  
POSIX共享内存获取
```

fd的方式

```
fd = shm_open(name, ...);使用
```

`mmap`映射到进程地址空间

```
addr = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

POSIX共享内存可以在无关的进程之间共享一个内存区域。和System V信号相比，POSIX使用了文件系统来标识共享内存，并且调用操作文件的接口来操作共享内存。每创建一个POSIX共享内存，挂载在`/dev/shm`下的`tmpfs`文件系统中就会新增一个文件。

和System V共享内存相比，POSIX共享内存的大小可以动态调整，因为POSIX共享内存是基于文件的，所以可以很方便地通过`ftruncate`函数来调整共享内存的大小。共享内存的使用者可以通过`munmap`和`mmap`重建映射。System V共享内存的大小在创建时就已经确定，无法再做调整。

总体来讲，POSIX共享内存要优于System V共享内存，建议使用POSIX共享内存。

11.5.1 共享内存的创建、使用和删除

共享内存的创建本质上是两个接口，首先是调用`shm_open`返回文件描述符，然后通过`mmap`将共享内存映射到进程的地址空间。两个函数的搭配很像System V的`shmget`函数和`shmat`函数。

`shm_open`函数的接口定义如下：

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

这里的`oflag`标志要包含`O_RDONLY`或`O_RDWR`标志位，除此以外，可以选择的标志位还有`O_CREAT`（表示创建）、`O_EXCL`（配合`O_CREAT`表示排他创建）。另外一个标志位是`O_TRUNC`，表示将共享内存的`size`截断成0。

`mode`参数可配合`O_CREAT`标志位使用，用于设定共享内存的访问权限。如果仅仅是打开共享内存，则可以传递0。`shm_open`总是需要`mode`参数。

`shm_open`函数调用成功时，会返回一个文件描述符。内核会自动设置`FD_CLOEXEC`标志位，即如果进程执行了`exec`函数，则该文件描述符会被自动关闭。

因为共享内存是文件，所以可以调用文件相关的函数，如`fstat`函数、`fchmod`函数和`fchown`函数。其中最重要常用的函数要属`ftruncate`函数。因为新创建的共享内存，默认大小总是0。所以在调用`mmap`之前，需要先调用`ftruncate`函数，以调整文件的大小。

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
```

调整了`size`之后，就可以调用`mmap`函数将共享内存映射到进程的地址空间了。对于其他参与通信的进程，可能需要调用`fstat`接口来获取共享内存区的大小。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int fstat(int fd, struct stat *buf);
```

通过该接口可以获取到共享内存的大小。

在`mmap`将共享内存映射到进程的地址空间之后，就可以通过操作内存来通信了。对这块内存的所有修改，其他进程都可以看到。

结束通信任务后，可以通过调用`munmap`函数解除映射。如果彻底不需要共享内存了，可以通过

`shm_unlink`函数来删除。该函数的接口定义如下：

```
int shm_unlink(const char *name);
```

删除一个共享内存对象，并不会影响既有的映射。内核维护有引用计数，当所有的进程都通过`munmap`解除映射之后，共享内存对象才会真正被删除。

如果不执行`shm_unlink`，共享内存对象中的数据则具有内核持久性。哪怕所有的进程都通过`munmap`解除了映射，只要不调用`shm_unlink`，其中的数据就不会丢失。当然，如果系统重启，那么其中的共享内存对象也就不复存在了。

11.5.2 共享内存与tmpfs

POSIX共享内存是建立在tmpfs基础之上的。事实上，System V共享内存也是建立在tmpfs基础上的。

从glibc的角度来看，shm_open的实现是非常简单的：

```
#define SHMDIR    (_PATH_DEV "shm/")
int
shm_open (const char *name, int oflag, mode_t mode)
{
    size_t namelen;
    char *fname;
    int fd;
    /* 滤除用户给出的名字中的一个或多个

/字符

*/
    while (name[0] == '/')
        ++name;
    if (name[0] == '\0')
    {
        /* The name "/" is not supported. */
        __set_errno (EINVAL);
        return -1;
    }
    /* 这一部分是生成完整的路径名

*/
    namelen = strlen (name);
    fname = (char *) __alloca (sizeof SHMDIR - 1 + namelen + 1);
    __memcpy (__memcpy (fname, SHMDIR, sizeof SHMDIR - 1),
              name, namelen + 1);
    fd = open (name, oflag, mode);
    if (fd != -1)
    {
        /* 给文件描述符设置

FD_CLOEXEC标志位

*/
        int flags = fcntl (fd, F_GETFD, 0);
        if (__builtin_expect (flags, 0) != -1)
        {
            flags |= FD_CLOEXEC;
            flags = fcntl (fd, F_SETFD, flags);
        }
        if (flags == -1)
        {
            /* Something went wrong. We cannot return the descriptor. */
            int save_errno = errno;
            close (fd);
            fd = -1;
            __set_errno (save_errno);
        }
    }
    return fd;
}
```

该函数就做了三件事：

1) 生成真正的文件名：当用户调用shm_open传递的文件名为name时，文件的全路径

是/dev/shm/name。

2) 创建或打开/dev/shm/name文件。

3) 给打开的文件设置FD_CLOEXEC标志位。

前文曾不断提及，mmap才是关键，无论是通过open获取到fd还是根据shm_open获取到fd，并没有什么本质的区别。看到glibc的shm_open实现后，我们更能够理解这个观点，的确没有本质区别，shm_open，不过就是open披了一个马甲。

接下来可以讲讲tmpfs相关的内容了。在shm_open的实现中选择/dev/shm这个路径并不是随意而为之的。glibc为了实现POSIX共享内存，需要将一个tmpfs挂载到/dev/shm这个路径下。

tmpfs是一个内存文件系统，该文件系统可将所有的文件内容保持在内存之中，而不会写入到磁盘等持久化的设备中。一旦umount或系统重启，tmpfs里的内容就会全部丢失。

内核的文档中Documentation/filesystems/tmpfs.txt中介绍了tmpfs的作用：

- 总是存在内核的内部挂载（internal mount），这个内部挂载并不依赖于CONFIG_TMPFS，哪怕CONFIG_TMPFS编译选项没有打开，也不会影响到该内部机制的存在。它的存在是为共享匿名映射和System V共享内存服务的。

- glibc自2.2版本以来，为了实现POSIX共享内存的功能，需要一个挂载点为/dev/shm的tmpfs。

从文档中可以看出，无论是POSIX信号量、System V信号量还是共享匿名映射都是建立在tmpfs的基础上的，其统一的视图如图11-24所示。

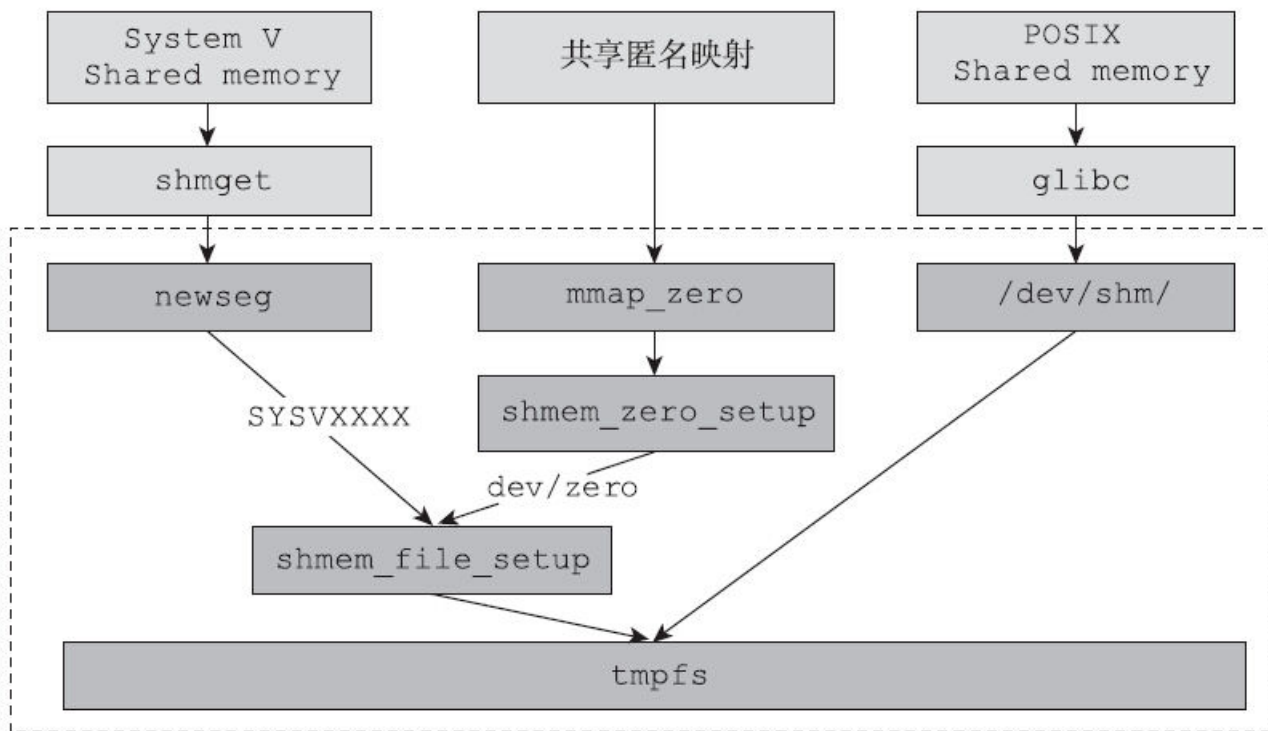


图11-24 共享内存与tmpfs

对于System V共享内存而言，其核心是tmpfs，外面封装了一层用来管理IPC的键值。当调用shmget创建System V共享内存时，会调用ipc/shm.c中的newseg函数。该函数会调用位于mm/shmem.c文件中的shmem_file_setup函数来创建一个与共享内存对应的struct file。代码如下所示：

```

sprintf (name, "SYSV%08x", key);
if (shmflg & SHM_HUGETLB) {
    /* hugetlb_file_setup applies strict accounting */
    if (shmflg & SHM_NORESERVE)
        acctflag = VM_NORESERVE;
    file = hugetlb_file_setup(name, size, acctflag,
        &shp->mlock_user, HUGETLB_SHMFS_INODE);
} else {
    if ((shmflg & SHM_NORESERVE) &&
        sysctl_overcommit_memory != OVERCOMMIT_NEVER)
        acctflag = VM_NORESERVE;
    file = shmem_file_setup(name, size, acctflag);
}

```

当调用shmat函数将System V共享内存attach到进程的地址空间时，内核会通过do_mmap函数，创建出基于该文件的共享映射，提供给用户使用。毫不意外，当用户调用shmdt函数解除映射时，内核会调用do_munmap。

在Linux实现中，传统的System V共享内存虽然没有显式地调用open-mmap-munmap这套流程，但是内在的核心逻辑是一致的。shmget获得了一个tmpfs的文件实例，shmat函数内部对应mmap，而shmdt函数内部对应munmap。

接下来分析共享匿名映射。创建共享匿名映射有两条路，其中一条就是打开/dev/zero文件，将获得的文件描述符fd传递给mmap函数。/dev/zero是一个特殊的文件，在drivers/char/mem.c中有如下内容：

```
static const struct memdev {
    const char *name;
    mode_t mode;
    const struct file_operations *fops;
    struct backing_dev_info *dev_info;
} devlist[] = {
    ...

    [5] = { "zero", 0666, &zero_fops, &zero_bdi },
    ...

}
static const struct file_operations zero_fops = {
    .llseek    = zero_llseek,
    .read      = read_zero,
    .write     = write_zero,
    .mmap      = mmap_zero,
};
```

如果打开/dev/zero文件，并将获得的文件描述符fd传给mmap系统调用，那么内核中mmap_region函数中调用的file->f_op->mmap函数，实质上调用的是mmap_zero函数，而mmap_zero函数，不过是shmem_zero_setup函数的简单封装。

```
static int mmap_zero(struct file *file, struct vm_area_struct *vma)
{
#ifdef CONFIG_MMU
    return -ENOSYS;
#endif
    if (vma->vm_flags & VM_SHARED)
        return shmem_zero_setup(vma);
    return 0;
}
```

创建共享匿名映射的另外一条路是调用mmap，传递-1作为fd的值。这种情况下也会走到shmem_zero_setup函数。请看mmap_region函数中的如下代码：

```
if (file) {
    ...
} else if (vm_flags & VM_SHARED) { /*共享匿名映射处理逻辑

*/
    error = shmem_zero_setup(vma);
    if (error)
        goto free_vma;
}
```

殊途同归，无论采用哪种方式创建共享匿名映射，最终都会调用到shmem_zero_setup函数。而该函数仅仅是shmem_file_setup的简单封装。

POSIX共享内存前面已经分析过了，通过挂载到/dev/shm路径下的tmpfs来实现内存的共享。glibc的shm_open用于创建一个文件，并且通过mmap映射到进程的地址空间。

从上面的讨论也可以看出，mmap和tmpfs是隐藏在共享内存背后的终极boss。无论是System V共享内存，还是POSIX共享内存，都摆脱不了tmpfs和mmap。区别仅仅是POSIX共享内存很直接，就是直接

在tmpfs下创建文件，直接通过mmap来使用内存区域，而System V共享内存穿了马甲，将tmpfs和mmap的相关操作隐藏到了内核中。

对tmpfs和共享内存感兴趣的话，《浅析Linux共享内存和tmpfs》^[1]是一篇不错的参考文档。

^[1] <http://hustcat.github.io/shared-memory-tmpfs/>。

第12章 网络通信：连接的建立

在互联网时代，网络通信编程已经是一个程序员必不可少的技能之一。几乎所有的产品都会涉及网络操作或访问。在Linux编程环境中，系统提供了socket套接字为程序员提供统一的网络编程接口。

本书将对socket套接字进行详细的分析，由于篇幅较多，所以将内容分为三章来讲述。本章主要讲解与连接相关的分析，包括socket、bind、connect、listen和accept系统调用及相关的源码追踪。这里假设读者有一定的Linux网络编程基础，所以对于系统调用的解释都是点到为止，只针对不常见或容易忽视的问题进行详细说明。

12.1 socket文件描述符

socket翻译成中文是插座、插槽的意思，而在网络编程中，其被翻译为“套接字”。Linux环境下，我们经常说“一切皆文件”。因此套接字也被视为一种文件描述符。首先，来看看如何使用socket系统调用创建一个套接字，代码如下：

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- 其中的参数解释如下。
- domain: 用于指示协议族名字，如AF_INET为IPv4。
 - type: 用于指示类型，如基于流通信的SOCK_STREAM。
 - protocol: 用于指示对于这种socket的具体协议类型。一般情况下，使用前两个参数限定后，只会存在一种协议类型对应该情况。这时，可以将protocol设置为0。但是在某些情况下，会存在多个协议类型，这时就必须指定具体的协议类型。

成功创建socket后，会返回一个文件描述符。失败时，该接口返回-1。

那么对于Linux内核来说，如何知道一个文件描述符是一个套接字，还是一个普通文件呢？其实这个问题也可以扩展到，内核如何知道一个文件描述符的具体类型，如何调用实际类型的操作函数呢？这仍然是VFS的魔力。

在第1章中，我们了解了文件描述符fd与内核文件结构struct file之间的关系，后者是内核用于管理文件的真正结构，其中的成员变量file->f_op为VFS支持的所有文件操作。VFS层无须关心该文件file的实际类型，它会直接调用file->f_op中的操作函数（这样的处理，与面向对象语言中的多态是类似的）。

对于套接字来说，只要在创建套接字时，将file->f_op设置为正确的套接字操作函数即可。该操作是在socket->sock_map_fd->sock_alloc_file中完成的，代码如下：

```
static int sock_alloc_file(struct socket *sock, struct file **f, int flags){
    .....

    /*      申请一个
    struct file, 并将
    socket_file_ops作为参数来传递。

    在
    alloc_file中, 会将
    socket_file_ops赋给
    file->f_op.

    */      file = alloc_file(&path, FMODE_READ | FMODE_WRITE,      &socket_file_ops);      .....

    /* 让
    sock->file指向
```


file, 完成

sock和

file的关联

```
*/    sock->file = file;    file->f_flags = O_RDWR | (flags & O_NONBLOCK);    file->f_pos = 0;    file->private_data = sock;    *f = file;    return fd;}
```

尽管Linux内核是使用C语言编写的，但是其应用了很多面向对象的设计思想。以这里的file为例，内核利用f_op（对象操作函数指针集合）指向具体对象的操作函数集合。这样一来，对于VFS来说，就只须关心struct file，而无须关心具体的对象类型了，它会在处理过程中，调用正确的处理函数。

12.2 绑定IP地址

在成功创建套接字后，该套接字仅仅是一个文件描述符，并没有任何地址与之关联。使用该socket发送数据包时，由于该socket没有任何IP地址，内核会根据策略自动选择一个地址。但是，在某些情况下，我们需要手工指定socket使用哪个IP地址进行发送。这时，就需要使用bind系统调用了。

12.2.1 bind的使用

bind系统调用的接口定义如下：

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

其中的参数解释如下。

- sockfd: 表示要绑定地址的套接字描述符。
- addr: 表示绑定到套接字的地址。
- addrlen: 表示绑定的地址长度。

返回值0表示成功，-1则表示错误。

因为Linux的套接字是针对多种协议族的，而每个协议族都可以有不同的地址类型。所以Linux套接字关于地址的系统调用，统一使用了一个公共结构体，并要求调用者将实际地址参数进行强制类型转换，以此来避免编译警告。

```
struct sockaddr {      sa_family_t sa_family;      char      sa_data[14];}
```

因为每个协议族的地址类型各不相同，所以需要通过参数addrlen来告诉内核这个地址的实际大小。



说明 struct sockaddr数据类型会在socket涉及地址的所有接口中出现。这是因为套接字接口要支持所有的协议族，所以涉及地址的地方都使用了一个统一的地址结构struct sockaddr。

下面是一个简单示例：

```
#include <stdlib.h>#include <stdio.h>#include <unistd.h>#include <sys/types.h>#include <sys/socket.h>#include <arpa/inet.h>#define LOOPBACK_ADDR 0x7F000001#define LISTEN_PORT
```

在上面的示例中，我们创建了一个TCP套接字，并将回环地址127.0.0.1和端口1234绑定到这个套接字上。运行这个程序，然后通过netstat检查监听端口：

```
[fgao@ubuntu ~]#netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp      0      0 127.0.0.1:1234 0.0.0.0:* LISTEN
```

从上面的输出可以看到，创建的套接字已经成功地绑定了指定的地址和端口。

12.2.2 bind的源码分析

bind源码入口位于net/socket.c中，如下所示：

```

SYSCTL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen){
    struct socket *sock;
    struct sockaddr_storage address;
    int err, fput_needed; /* 由文件描述符 */

    struct socket. 前文已经介绍了

    fd与

    struct socket的关联关系。

    /* sock = sockfd_lookup_light(fd, &err, &fput_needed); if (sock) { /* umyaddr是用户空间地址, 这里将其复制到内核空间 */

    address变量中

    /* err = move_addr_to_kernel(umyaddr, addrlen, (struct sockaddr *)&address); if (err >= 0) { /* 对

    bind动作进行安全性检查

    /* err = security_socket_bind(sock, (struct sockaddr *)&address, addrlen); if (!err) { /* 调用对应协议的

    bind动作

    /* err = sock->ops->bind(sock, (struct sockaddr *)&address, addrlen); } } fput_light(sock->

```

在bind的调用中，根据不同的协议调用不同的实现函数（Linux的内核代码中，大量使用了这种面向对象的设计思路）。对于AF_INET协议族来说，无论是面向连接的SOCK_STREAM类型，还是SOCK_DGRAM协议类型，其实现函数均是inet_bind。下面来看一下inet_bind的具体实现：

```

int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len){    struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;    struct sock *sk = sock->sk;    struct in

    如果具体协议实现了

bind函数, 则调用协议的

bind函数.

AF_INET协议族中, 只有

IPPROTO_ICMP和

IPPROTO_IP实现了自己的

bind函数.

IPPROTO_TCP和

IPPROTO_UDP都使用

AF_INET通用的函数, 即

这个

inet_bind.

    /*    if (sk->sk_prot->bind) {        err = sk->sk_prot->bind(sk, uaddr, addr_len);        goto out;}    err = -EINVAL;    /* 检查地址长度

    /*    if (addr_len < sizeof(struct sockaddr_in))        goto out;    if (addr->sin_family != AF_INET) {        /* 本来要求地址的协议族要与

sock相同, 必须为

```

AF_INET，但是这里有个兼容性问题，允许协

议族为

AF_UNSPEC并且地址为

INADDR_ANY的任意地址

```
*/          err = -EAFNOSUPPORT;          if (addr->sin_family != AF_UNSPEC ||          addr->sin_addr.s_addr != htonl(INADDR_ANY))          goto out;      }      /* 判断地址类型

*/      chk_addr_ret = inet_addr_type(sock_net(sk), addr->sin_addr.s_addr);      err = -EADDRNOTAVAIL;      /*
sysctl_ip_nonlocal_bind系统控制开关，允许
```

bind非本地

IP; inet->freebind为一个

socket选项，允许该

socket bind任意

IP; 在上面这些变量均不成立时，指定地址又不是任意的

本地地址

INADDR_ANY，地址类型又不是本地地址类型，多播或广播时，则

bind失败。

```
*/      if (!sysctl_ip_nonlocal_bind &&          !(inet->freebind || inet->transparent) &&          addr->sin_addr.s_addr != htonl(INADDR_ANY) &&          chk_addr_ret != RTN_LOCAL &&
```

PROT_SOCK(1024)，则需要检查用户是否有权限制建知名端口

```
*/      if (snum && snum < PROT_SOCK && !capable(CAP_NET_BIND_SERVICE))          goto out;      lock_sock(sk);      err = -EINVAL;      /* 确保套接字不会被
```

bind两次

```
*/      if (sk->sk_state != TCP_CLOSE || inet->inet_num)          goto out_release_sock;      /* 使用参数设置套接字的接收和发送地址
```

```
*/      inet->inet_rcv_saddr = inet->inet_saddr = addr->sin_addr.s_addr;      /* 如果参数地址是多播或广播类型，则重置发送源地址为
```

0，表示在发送时，使用的是设备地址

```
*/      if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)          inet->inet_saddr = 0;      /* Use device */      /* 调用协议自定义的操作函数
```

get_port，判断该端口是否可以。

虽然这里是一个查询的动作，但是却会有修改的动作。

当该端口可以使用时，会让

inet_sk(sk)->inet_num = snum; 这样做，是因为查询动作已经获得了锁。在确定可以使用该端口时，直接修

改

inet_num，这样既可以保证设置端口的原子性，同时还可以提高性能

```
*/      if (sk->sk_prot->get_port(sk, snum)) {          inet->inet_saddr = inet->inet_rcv_saddr = 0;          err = -EADDRINUSE;          goto out_release_sock;      }      /* 如果设置了
```

bind地址，则置上相应的标志

```
*/    if (inet->inet_rcv_saddr)        sk->sk_userlocks |= SOCK_BINDADDR_LOCK;    /* 如果设置了源端口，则设置相应的标志
```

```
*/    if (snum)        sk->sk_userlocks |= SOCK_BINDPORT_LOCK;    /* 设置
```

inet_sport, 其为网络序

```
*/    inet->inet_sport = htons(inet->inet_num);    /* 重置目的地址和端口
```

```
*/    inet->inet_daddr = 0;    inet->inet_dport = 0;    /* 重置该套接字的路由信息
```

```
*/    sk_dst_reset(sk);    err = 0; out_release_sock:    release_sock(sk); out:    return err; }
```

无论是APUE还是man手册，在讲解bind的时候都有点问题，或有偏差，或不够详尽。从上面的源码我们知道，通过使用系统控制开关sysctl_ip_nonlocal_bind或套接字选项可以让套接字bind一个非本机地址。但APUE却说套接字只能绑定本机的有效地址——当然这也是由于APUE距现在的时间太久了，而man手册都没有提及非本机地址的事情。

12.3 客户端连接过程

12.3.1 connect的使用

connect的原型为：

```
#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

其中的参数解释如下：

- int sockfd: 套接字描述符。
- const struct sockaddr*addr: 要连接的地址。
- socklen_t addrlen: 要连接的地址长度。

返回值0表示成功，-1表示失败。

connect的用途是使用指定的套接字去连接指定的地址。对于面向连接的协议（套接字类型为SOCK_STREAM），connect只能成功一次（当然要如此，因为真正的连接已经建立了）。如果重复调用connect，会返回-1表示失败，同时错误码为EISCONN。而对于非面向连接的协议（套接字类型为SOCK_DGRAM），则可以执行多次connect（因为这时的connect仅仅是设置了默认的目的地址）。

对于TCP套接字来说，connect实际上是要真正地进行三次握手，所以其默认是一个阻塞操作。那么是否可以写一个非阻塞的TCP connect代码呢？这是一个合格的网络开发工程师的基本功，具体的实现可以参看UNPv1的实现。更重要是要理解其原理，这样才能在需要的时候，信手拈来。

12.3.2 connect的源码分析

connect的源码入口位于socket.c，代码如下：

```
SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __user *, uservaddr, int, addrlen){ struct socket *sock; struct sockaddr_storage address; int err, fput_needed;

struct socket */ sock = sockfd_lookup_light(fd, &err, &fput_needed); if (!sock) goto out; /* 将用户空间地址复制到内核空间变量

address中

*/ err = move_addr_to_kernel(uservaddr, addrlen, (struct sockaddr *)&address); if (err < 0) goto out_put; /* 安全性检查

*/ err = security_socket_connect(sock, (struct sockaddr *)&address, addrlen); if (err) goto out_put; /* 与

bind类似，调用与协议族对应的

connect操作函数

*/ err = sock->ops->connect(sock, (struct sockaddr *)&address, addrlen, sock->file->f_flags);out_put: fput_light(sock->file, fput_needed);out: return err
```

对于AF_INET协议族来说，面向连接的协议类型是SOCK_STREAM，其连接函数为inet_stream_connect，而非面向连接的协议类型SOCK_DGRAM，其连接函数为inet_dgram_connect。这很合理，因为从connect的功能实现上看，两者的实现效果完全不同。

让我们先从简单的inet_dgram_connect入手。

```
int inet_dgram_connect(struct socket *sock, struct sockaddr * uaddr, int addr_len, int flags){ struct sock *sk = sock->sk; /* 长度合法性检查

*/ if (addr_len < sizeof(uaddr->sa_family)) return -EINVAL; /* 如果协议族为

AF_UNSPPEC，则先执行

disconnect */ if (uaddr->sa_family == AF_UNSPPEC) return sk->sk_prot->disconnect(sk, flags); /* 如果该套接字没有指定源端口，并且系统自动绑定端口失败，则返回错误

*/ if (!inet_sk(sk)->inet_num && inet_autobind(sk)) return -EAGAIN; /* 调用具体协议的

connect实现函数

*/ return sk->sk_prot->connect(sk, (struct sockaddr *)uaddr, addr_len);}
```

udp_prot是UDP协议中所有自定义操作函数的集合。其connect的实现函数为ip4_datagram_connect。

```
int ip4_datagram_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len){ struct inet_sock *inet = inet_sk(sk); struct sockaddr_in *usin = (struct sockaddr_in *) u

*/ if (addr_len < sizeof(*usin)) return -EINVAL; /* 检查是否为

AF_INET协议族

*/ if (usin->sin_family != AF_INET) return -EAFNOSUPPORT; /* 因为

connect会改变目的地址，所有

socket中保存的路由缓存已经无用，必须重置。

*/ sk_dst_reset(sk); lock_sock(sk); /* 得到套接字绑定的发送接口

*/ oif = sk->sk_bound_dev_if; saddr = inet->inet_saddr; /* 在目的地址是多播地址的情况下。

如果该套接字没有绑定网卡，则出口网卡为设置的多播网卡索引；

如果该套接字没有绑定源
```


IP, 则使用设置的多播源地址:

```
/*      if (ipv4_is_multicast(usin->sin_addr.s_addr)) {          if (!oif)              oif = inet->mc_index;          if (!saddr)              saddr = inet->mc_addr;          }      /* 判断设置的

*/      fl4 = &inet->cork.fl.u.ip4;      rt = ip_route_connect(fl4, usin->sin_addr.s_addr, saddr,          RT_CONN_FLAGS(sk), oif,          sk->sk_protocol,

*/      if ((rt->rt_flags & RTCF_BROADCAST) && !sock_flag(sk, SOCK_BROADCAST)) {          ip_rt_put(rt);          err = -EACCES;          goto out;          }      /* 如果套接字没有设置发送地址或接收地址, 则使用

*/      if (!inet->inet_saddr)          inet->inet_saddr = fl4->saddr; /* Update source address */      if (!inet->inet_rcv_saddr) {          inet->inet_rcv_saddr = fl4->saddr;          if

*/      inet->inet_daddr = fl4->daddr;      inet->inet_dport = usin->sin_port;      sk->sk_state = TCP_ESTABLISHED;      inet->inet_id = jiffies;      /* 重新设置路由信息

*/      sk_dst_set(sk, &rt->dst);      err = 0;out:      release_sock(sk);      return err;}
```

由于功能比较简单, 所以UDP的connect实现源码也一目了然, 可以看到, 只是设置了目的IP、端口和路由信息。下面对比一下TCP的connect实现, 其实现比UDP要复杂得多, 代码如下:

```
int inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,          int addr_len, int flags){      /*      从

socket结构获得

sock结构, 后者是内核真正用于管理网络层的套接

字结构

*/      struct sock *sk = sock->sk;      int err;      long timeo;      /* 地址长度检查

*/      if (addr_len < sizeof(uaddr->sa_family))          return -EINVAL;      lock_sock(sk);      /* 对

AF_UNSPEC兼容性处理

*/      if (uaddr->sa_family == AF_UNSPEC) {          err = sk->sk_prot->disconnect(sk, flags);          sock->state = err ? SS_DISCONNECTING : SS_UNCONNECTED;          goto out;          }

STREAM协议是有连接状态的, 所以需要对接接字进行状态检查

*/      switch (sock->state) {          default:              err = -EINVAL;              goto out;          /* 若连接已经建立, 则返回错误

*/          case SS_CONNECTED:              err = -EISCONN;              goto out;          /* 若连接正在进行中, 则返回错误

*/          case SS_CONNECTING:              err = -EALREADY;              /* Fall out of switch with err, set for this state */              break;          /* 当前为未连接状态

*/          case SS_UNCONNECTED:              err = -EISCONN;              /* sock的状态是未连接, 但是套接字的

sk_state却不是关闭状态。

此时无法进行连接

*/              if (sk->sk_state != TCP_CLOSE)                  goto out;          /*              既然需要产生连接, 那么每种具体的协议肯定都有自己的实现。

所以这会调用具体协议的实现函数。

*/              err = sk->sk_prot->connect(sk, uaddr, addr_len);              if (err < 0)                  goto out;          /* 将

sock状态更改为正在连接中

*/              sock->state = SS_CONNECTING;          /* Just entered SS_CONNECTING state; the only          * difference is that return value in non-blocking          * case is EINPROGRE

EINPROGRESS, 表示正在连接中。

*/
```

当

connect为非阻塞时，就会返回这个错误

```
*/      err = -EINPROGRESS;      break;    }    /*      检查是否需要连接超时。若设置了非阻塞标志，则
```

timeo为假。

若设置了阻塞标志，则

timeo为真。

```
*/      timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);    /* 当前连接状态为正在连接的状态（即刚发送了
```

syn或收到

syn)

```
*/      if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV)){    /* 如果没有超时标志或连接超时或失败，则返回
```

```
*/      if (!timeo || !inet_wait_for_connect(sk, timeo))      goto out;    /* 判断是否由于信号导致等待连接退出
```

```
*/      err = sock_intr_errno(timeo);    /* 如果有未处理的信号，则返回失败，表示信号中断了连接
```

```
*/      if (signal_pending(current))      goto out;    }    /* 连接被关闭了。原因可能是对端
```

RST，超时等

```
*/      if (sk->sk_state == TCP_CLOSE)      goto sock_error;    /* 至此，连接成功
```

```
*/      sock->state = SS_CONNECTED;      err = 0;out:      release_sock(sk);return err;sock_error:      err = sock_error(sk) ? : -ECONNABORTED;      sock->state = SS_UNCONNECTED;      if (sk-
```

接下来，就需要进入TCP协议自定义的connect函数tcp_v4_connect了，代码如下：

tcp_v4_connect了，代码如下：

```
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len){      struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;      struct inet_sock *inet = inet_sk(sk);
```

```
*/      if (addr_len < sizeof(struct sockaddr_in))      return -EINVAL;    /* 协议族类型检查
```

```
*/      if (usin->sin_family != AF_INET)      return -EAFNOSUPPORT;    /* 设置下一跳和目的地址
```

```
*/      nexthop = daddr = usin->sin_addr.s_addr;    /* 获得
```

IP选项

```
*/      inet_opt = rcu_dereference_protected(inet->inet_opt,      sock_owned_by_user(sk));    /* 如果有严格源路由的
```

IP选项

```
*/      if (inet_opt && inet_opt->opt.srr) {    /* 若地址为
```

0，则返回错误

```
*/      if (!daddr)      return -EINVAL;    /* 因为严格源路由的
```

IP选项，所以下一跳要设置为选项中的第一跳地址

```
*/      nexthop = inet_opt->opt.faddr;    }    /* 设置源端口和目的端口
```

```

*/   orig_sport = inet->inet_sport;   orig_dport = usin->sin_port;   fl4 = &inet->cork.fl.u.ip4;   /* 查找路由

*/   rt = ip_route_connect(fl4, nexthop, inet->inet_saddr, RT_CONN_FLAGS(sk), sk->sk_bound_dev_if, IPPROTO_TCP, orig_sport, orig_

*/   if (IS_ERR(rt)) {   err = PTR_ERR(rt);   if (err == -ENETUNREACH)   IP_INC_STATS_BH(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);   return err;   }   /* 如

*/   if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {   ip_rt_put(rt);   return -ENETUNREACH;   }   /* 如果没有

```

IP选项或没有设置严格路由，那么目的地址即为路由结果的目的地址

```

*/   if (!inet_opt || !inet_opt->opt.srr)   daddr = fl4->daddr;   /* 如果没有设置源地址，则使用路由结果的源地址

*/   if (!inet->inet_saddr)   inet->inet_saddr = fl4->saddr;   /* 套接字的接收地址即为源地址

*/   inet->inet_rcv_saddr = inet->inet_saddr;   /*   若保存的

```

TCP选项有时间戳，并且目的地址与要连接的地址不同，

则需要重置时间戳及相关变量

```

*/   if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {   /* Reset inherited state */   tp->rx_opt.ts_recent   = 0;   tp->rx_opt.ts_recent_stam

   戳信息，则尝试从对端

```

peer中获得时间戳信息

```

*/   if (tcp_death_row.sysctl_tw_recycle && !tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr) {   struct inet_peer *peer = rt_get_peer(rt, fl4->daddr);   /* 如

*/   if (peer) {   inet_peer_refcheck(peer);   /* 如果对端保存的时间戳信息还没有过期

*/   if ((u32)get_seconds() - peer->tcp_ts_stamp <= TCP_PAWS_MSL) {   /*   利用对端保存的时间戳信息初始化当前套接字的时间戳选项

*/   tp->rx_opt.ts_recent_stamp = peer->tcp_ts_stamp;   tp->rx_opt.ts_recent = peer->tcp_ts;   }   }   }   /* 设置目的端口和地

*/   inet->inet_dport = usin->sin_port;   inet->inet_daddr = daddr;   /* 设置

```

IP头的选项长度

```

*/   inet_csk(sk)->icsk_ext_hdr_len = 0;   if (inet_opt)   inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;   /* 初始化

```

MSS */ tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT; /* 设置

TCP的状态为

SYN_SENT. 即发送了

syn包

```

*/   tcp_set_state(sk, TCP_SYN_SENT);   /* 将套接字加入到

```

hash表中，并分配源端口

```

*/   err = inet_hash_connect(&tcp_death_row, sk);   if (err)   goto failure;   /* 检查源端口或目的端口是否发生了变化，如果发生了变化则重新查找路由

```

```

*/   rt = ip_route_newports(fl4, rt, orig_sport, orig_dport, inet->inet_sport, inet->inet_dport, sk);   if (IS_ERR(rt)) {   err = PTR_ERR(rt);   rt

```

GSO功能了

```

*/   sk->sk_gso_type = SKB_GSO_TCPV4;   sk_setup_caps(sk, &rt->dst);   /* 如果没有设置初始的序列号，则根据双方地址，随机生成端口

```

```

    /*      if (!tp->write_seq)          tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,

inet_id. 该

ID用于生成

IP报文的

ID值

    /*      inet->inet_id = tp->write_seq ^ jiffies;      /* 一切准备工作完毕,

tcp_connect生成

SYN报文并发送

    /*      err = tcp_connect(sk);      rt = NULL;      if (err)          goto failure;      return 0;failure:      /*      * This unhashes the socket and releases the local port,      * if necessa

```

下面来分析tcp_connect，看看内核是如何发送SYN包的，代码如下：

```

int tcp_connect(struct sock *sk){      struct tcp_sock *tp = tcp_sk(sk);      struct sk_buff *buff;      int err;      /* 初始化

TCP连接控制块

    /*      tcp_connect_init(sk);      /* 申请报文内存

    /*      buff = alloc_skb_fclone(MAX_TCP_HEADER + 15, sk->sk_allocation);      if (unlikely(buff == NULL))          return -ENOMEM;      /* 目前我们不知道后面会填充哪些

TCP选项，所以直接在

skb的首部保

留

TCP协议的最大长度，从而保证了足够的空间，避免重新分配内存。

    /*      skb_reserve(buff, MAX_TCP_HEADER);      tp->snd_nxt = tp->write_seq;      /* 初始化报文

    /*      tcp_init_nodata_skb(buff, tp->write_seq++, TCPCRDR_SYN);      TCP_ECN_send_syn(sk, buff);      /* 设置

TCP控制块相关的发送变量，并发送该

SYN报文

    /*      TCP_SKB_CB(buff)->when = tcp_time_stamp;      tp->retrans_stamp = TCP_SKB_CB(buff)->when;      skb_header_release(buff);      __tcp_add_write_queue_tail(sk, buff);      sk->sk_w

TCP套接字对应的重传定时器

    /*      inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,      inet_csk(sk)->icsk_rto, TCP_RTO_MAX);      return 0;}

```

至此，TCP的连接过程已经分析完毕，其中涉及的某些过程会在后面进行具体分析。

12.4 服务器端连接过程

12.4.1 listen的使用

服务器端用**listen**来监听端口，其原型为：

```
#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

其中的参数解释如下：

- 参数**int sockfd**：成功创建的TCP套接字。

- int backlog**：定义TCP未处理连接的队列长度。该队列虽然已经完成了三次握手，但服务器端还没有执行**accept**的连接。APUE中说，**backlog**只是一个提示，具体的数值实际上是由系统来决定的。后面会通过学习内核源码来确定这一点。

函数的返回值为0，表示成功；-1表示失败。

12.4.2 listen的源码分析

listen的源码入口位于socket.c，代码如下：

```
SYSCALL_DEFINE2(listen, int, fd, int, backlog){    struct socket *sock;    int err, fput_needed;    int somaxconn;    /* 从文件描述符得到

socket结构

*/    sock = sockfd_lookup_light(fd, &err, &fput_needed);    if (sock) {        /* 得到系统设置的最大未处理连接队列长度

*/        somaxconn = sock_net(sock->sk)->core.sysctl_somaxconn;        /* 如果用户指定的参数

backlog大于系统最大值，则使用系统最大值

*/        if ((unsigned)backlog > somaxconn)backlog = somaxconn;        /* 进行安全性检查

*/        err = security_socket_listen(sock, backlog);        /* 通过检查后，就调用指定协议族的

listen实现函数

*/        if (!err)            err = sock->ops->listen(sock, backlog);        fput_light(sock->file, fput_needed);    }    return err;}
```

AF_INET协议族的listen实现函数为inet_listen，代码如下：

```
int inet_listen(struct socket *sock, int backlog){    struct sock *sk = sock->sk;    unsigned char old_state;    int err;    lock_sock(sk);    err = -EINVAL;    /* 如果套接字状态不是4

*/    if (sock->state != SS_UNCONNECTED || sock->type != SOCK_STREAM)        goto out;    /* 得到之前的

TCP连接状态

*/    old_state = sk->sk_state;    /* 如果之前的状态不是关闭或监听，则返回错误

*/    if (!(1 << old_state) & (TCPF_CLOSE | TCPF_LISTEN)))        goto out;    /* 经过前面的状态过滤，这里只可能是关闭或监听状态。

如果当前已经是监听状态了，那么我们只须改变

backlog的值：

如果是关闭状态，则需要真正地启动监听操作。

*/    if (old_state != TCP_LISTEN) {        err = inet_csk_listen_start(sk, backlog);        if (err)            goto out;    }    /* 更新

backlog的值

*/    sk->sk_max_ack_backlog = backlog;    err = 0;out:    release_sock(sk);    return err;}
```

接下来进入inet_csk_listen_start，代码如下：

```
int inet_csk_listen_start(struct sock *sk, const int nr_table_entries){    struct inet_sock *inet = inet_sk(sk);    struct inet_connection_sock *icsk = inet_csk(sk);    /* 为连接

*/    int rc = reqsk_queue_alloc(&icsk->icsk_accept_queue, nr_table_entries);    if (rc != 0)        return rc;    /* 初始化工作

*/    sk->sk_max_ack_backlog = 0;    sk->sk_ack_backlog = 0;    inet_csk_delack_init(sk);    /* 虽然这里是先将连接的状态设为了监听状态，看似有一个竞争时间窗口，但实际上只有在

get_port

成功以后，该套接字才被加入到哈希表中--从系统的角度看，套接字加入到哈希表中后，才会真正

处于监听状态，可以接受连接请求了。因此实际上并没有竞争发生

*/    sk->sk_state = TCP_LISTEN;    /* 使用
```

get_port进行端口绑定

```
*/    if (!sk->sk_prot->get_port(sk, inet->inet_num)) {          /* 设置源端口

*/

    inet->inet_sport = htons(inet->inet_num);          /* 清除路由缓存

*/

    sk_dst_reset(sk);          /* 将套接字加入到哈希表中，这时才可以接受新连接

*/sk->sk_prot->hash(sk);          return 0;    }          /* 绑定端口失败，则设置连接未关闭状态

*/

    sk->sk_state = TCP_CLOSE;          /* 释放连接请求队列空间

*/

    __reqsk_queue_destroy(&icsk->icsk_accept_queue);    return -EADDRINUSE;}
```

现在服务器端已经处于监听状态，可以接收客户端的连接请求了。同时，通过源码跟踪，也可以发现在第二个参数不超过系统限制的最大值的情况下，内核已直接使用其值作为已连接队列的长度了。

12.4.3 accept的使用

`accept`用于从指定套接字的连接队列中取出第一个连接，并返回一个新的套接字用于与客户端进行通信，示例代码如下：

```
#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int accept4(int sockfd, struct sockaddr *addr,
socklen_t *addrlen, int flags);
```

其中的参数解释如下：

·`int sockfd`：处于监听状态的套接字。

·`struct sockaddr*addr`：用于保存对端的地址信息。

·`socklen_t*addrlen`：是一个输入输出值。调用者将其初始化为`addr`缓存的大小，`accept`返回时，会将其设置为`addr`的大小。

·`int flags`：是新引入的系统调用`accept4`的标志位；目前支持`SOCK_NONBLOCK`和`SOCK_CLOEXEC`。

关于返回值，若执行成功，则返回一个非负的文件描述符；若失败则返回-1。



注意 若不关心对端地址信息，则可以将`addr`和`addrlen`设置为`NULL`。

12.4.4 accept的源码分析

accept的源码入口位于文件socket.c，代码如下：

```
SYSCALL_DEFINE3(accept, int, fd, struct sockaddr __user *, upeer_sockaddr, int __user *, upeer_addrlen){ return sys_accept4(fd, upeer_sockaddr, upeer_addrlen, 0);}
```

进入sys_accept4，代码如下：

```
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr, int __user *, upeer_addrlen, int, flags){ struct socket *sock, *newsock; struct file *f;

/* if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK)) return -EINVAL;  /* 保证设置的非阻塞标志

SOCK_NONBLOCK与

O_NONBLOCK相同

*/ if (SOCK_NONBLOCK != O_NONBLOCK && (flags & SOCK_NONBLOCK)) flags = (flags & ~SOCK_NONBLOCK) | O_NONBLOCK;  /* 通过文件描述符获得

socket结构

*/ sock = sockfd_lookup_light(fd, &err, &fput_needed); if (!sock) goto out; err = -ENFILE;  /* 申请一个新的

socket结构

*/ newsock = sock_alloc(); if (!newsock) goto out_put;  /* 新的

socket的类型和操作函数与监听

socket一致

*/ newsock->type = sock->type; newsock->ops = sock->ops;
/* 这里必须增加该套接字模块的引用计数。这是因为这个套接字模块可能不是

Linux内核内置的。

为了保证在套接字的使用过程中，该模块不会被意外卸载，所以，在创建套接字时，需要增加相应

的模块计数

*/ __module_get(newsock->ops->owner);  /* 为新的

socket类型，申请一个新的文件描述符

*/ newfd = sock_alloc_file(newsock, &newfile, flags); if (unlikely(newfd < 0)) { err = newfd; sock_release(newsock); goto out_put; }  /* 对

accept操作进行安全性检查

*/ err = security_socket_accept(sock, newsock); if (err) goto out_fd;  /* 执行协议族的

accept操作函数

*/ err = sock->ops->accept(sock, newsock, sock->file->f_flags); if (err < 0) goto out_fd;  /* 用户想获得对端地址

*/ if (upeer_sockaddr) {  /* 获得对端地址

*/ if (newsock->ops->getname(newsock, (struct sockaddr *)&address, &len, 2) < 0) { err = -ECONNABORTED; goto out_fd; }

*/ err = move_addr_to_user((struct sockaddr *)&address, len, upeer_sockaddr, upeer_addrlen); if (err < 0) goto out_fd; }  /* 将文件描述符

newfd和文件管理结构
```

newfile安装到文件表中

```
*/fd_install(newfd, newfile); /* 此时，已保证
```

accept成功执行，将

newfd赋给

err，并在后面返回

```
err /* err = newfd;out_put: fput_light(sock->file, fput_needed);out: return err;out_fd: fput(newfile); put_unused_fd(newfd); goto out_put;}
```

对于AF_INET协议族，accept的实现函数为inet_accept，代码如下：

```
int inet_accept(struct socket *sock, struct socket *newsock, int flags){ struct sock *sk1 = sock->sk; int err = -EINVAL; /* 调用具体协议的
```

accept操作，并得到新的

sock结构

```
*/ struct sock *sk2 = sk1->sk_prot->accept(sk1, flags, &err); if (!sk2) goto do_err; /* 锁住新的
```

```
sk2 /* lock_sock(sk2); /* 记录
```

RFS信息

```
*/ sock_rps_record_flow(sk2); WARN_ON(!(1 << sk2->sk_state) & (TCPF_ESTABLISHED | TCPF_CLOSE_WAIT | TCPF_CLOSE))); /* 将新的
```

sock与调用者传递的

socket关联起来

```
*/ sock_graft(sk2, newsock); /* 设置
```

socket为连接状态

```
*/ newsock->state = SS_CONNECTED; err = 0; /* 释放
```

sk2的控制权

```
*/ release_sock(sk2); do_err: return err;}
```

对于TCP协议来说，其accept实现函数如下：

```
struct sock *inet_csk_accept(struct sock *sk, int flags, int *err){ struct inet_connection_sock *icsk = inet_csk(sk); struct sock *newsk; int error; /* 获得
```

sk的控制权

```
*/ lock_sock(sk); error = -EINVAL; /* sk. 即
```

TCP连接，若不是监听状态则报错

```
*/ if (sk->sk_state != TCP_LISTEN) goto out_err; if (reqsk_queue_empty(&icsk->icsk_accept_queue)) { /* 已连接队列为空
```

```
*/ /* 得到
```

sk的超时时间

```
*/ long timeo = sock_rcvtimeo(sk, flags & O_NONBLOCK); /* If this is a non blocking socket don't sleep */ error = -EAGAIN; /* 如果超时为
```

0，即非阻塞，则报错退出

```
*/      if (!timeo)          goto out_err;      /* 以
```

timeo为超时时间，等待一个新的连接

```
*/      error = inet_csk_wait_for_connect(sk, timeo);      if (error)          goto out_err;      }      /* 得到新的
```

```
sock */      newsk = reqsk_queue_get_child(&icsk->icsk_accept_queue, sk);      WARN_ON(newsk->sk_state == TCP_SYN_RECV);out:      /* 释放
```

sk的控制权，并返回新建连接

```
newsk */      release_sock(sk);      return newsk;out_err:      newsk = NULL;      *err = error;      goto out;}
```

12.5 TCP三次握手的实现分析

前面两节分别从客户端和服务端的角度，来分析和学习TCP的连接过程。本节将从TCP三次握手的数据包交互过程，来研究TCP连接的建立。如果不熟悉TCP握手的三个数据包，则请自行阅读相关材料。

三次握手的过程如图12-1所示。

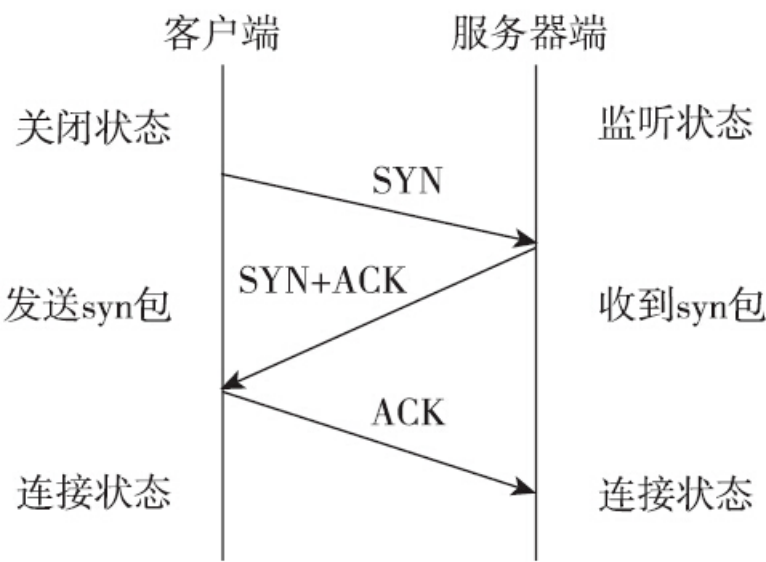


图12-1 TCP三次握手的过程

12.5.1 SYN包的发送

SYN包是指客户端主动建立一个TCP新连接的第一个包，其TCP标志为SYN，表示同步TCP的序列号。

SYN包的发送是在tcp_connect函数中完成的，下面对SYN包的构建做进一步分析。在tcp_connect函数中，通过调用tcp_init_nodata_skb（buff，tp->write_seq++，TCPHDR_SYN）来完成SYN包的构建，示例代码如下：

```
static void tcp_init_nodata_skb(struct sk_buff *skb, u32 seq, u8 flags){ /* 设置为

CHECKSUM_PARTIAL，表示需要计算

TCP校验和

*/      skb->ip_summed = CHECKSUM_PARTIAL; /* 初始化校验和信息

*/      skb->csum = 0; /* TCP_SKB_CB是一个宏，用于将

skb->cb转换为

TCP的控制块

*/      /* 设置

TCP首部标志位

*/      TCP_SKB_CB(skb)->tcp_flags = flags; /* 重置控制块的

SACK标志位

*/      TCP_SKB_CB(skb)->sacked = 0; /* 初始化

skb的

GSO */      skb_shinfo(skb)->gso_segs = 1;      skb_shinfo(skb)->gso_size = 0;      skb_shinfo(skb)->gso_type = 0; /* 设置

TCP的序列号

*/      TCP_SKB_CB(skb)->seq = seq; /* 如果是

SYN或

FIN包，则增加

TCP序列号

*/      if (flags & (TCPHDR_SYN | TCPHDR_FIN))          seq++; /* 设置结束序列号

*/      TCP_SKB_CB(skb)->end_seq = seq;}
```

从源码中可以发现，这个函数只是设置TCP控制块的序列号和标志，并没有真正构建TCP数据包。那么，让我们回过头来看tcp_connect，但是在tcp_init_nodata_skb和tcp_transmit_skb之间再没有任何与构建数据包相关的代码了。那么也只剩下一个可能，即在tcp_transmit_skb中实现数据包的构建，这样也合乎道理。tcp_transmit_skb作为TCP发送函数的入口，统一实现了TCP数据包的构建。

下面是其中用于构造TCP数据包的相关代码：

```
/* 为

TCP报文头部保存空间

*/skb_push(skb, tcp_header_size);/* 重置
```

TCP报文头指针

```
*/skb_reset_transport_header(skb);/* 设置
```

skb的所有者为

sk, 同时增加

sk的写缓存的使用统计

```
*/skb_set_owner_w(skb, sk);/* 得到
```

TCP报文头的内存指针, 开始构建

TCP的报文头部

```
*//* 这里设置了源端口、目的端口、序列号、确认序列号、包长及标志位
```

```
*/th = tcp_hdr(skb);th->source = inet->inet_sport;th->dest = inet->inet_dport;th->seq = htonl(tcb->seq);th->ack_seq = htonl(tp->rcv_nxt);*((__
```

SYN包, 则

TCP窗口不会被扩展

```
*/if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) { /* RFC1323: The window in SYN & SYN/ACK segments * is never scaled. */ th->window = htons(min(tp->rcv_wnd, 65535U));}
```

TCP的校验和与紧急指针为

```
0 */th->check = 0;th->urg_ptr = 0;/* 检查是否需要设置紧急指针
```

```
*/if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) { if (before(tp->snd_up, tcb->seq + 0x10000)) { th->urg_ptr = htons(tp->snd_up - tcb->seq); th-
```

TCP选项部分

```
*/tcp_options_write((__be32 *) (th + 1), tp, &opts);/* 计算
```

TCP的校验和

```
*/icsk->icsk_af_ops->send_check(sk, skb);/* 完成了
```

TCP数据包的构建, 将数据包交给

IP层

```
*/err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);
```

从上面的代码中, 我们可以进一步领会Linux内核协议栈的数据包传输机制——每一层都专注于自己的工作。对于TCP传输层来说, 只须负责在skb中构建自己的首部, 然后将skb数据包传递给IP层做进一步的处理即可。

12.5.2 接收SYN包，发送SYN+ACK包

为了跟踪SYN包的接收流程，首先进入内核并接收发给本机数据包的入口`ip_local_deliver_finish`，代码如下：

```
static int ip_local_deliver_finish(struct sk_buff *skb) {    /* 得到设备的网络空间

    /*      struct net *net = dev_net(skb->dev);    /* 取走网络层报文头部

    /*      __skb_pull(skb, ip_hdrlen(skb));    /* 重置传输层报文头部

    /*      skb_reset_transport_header(skb);    rcu_read_lock();    {    /* 得到传输层协议

    /*      int protocol = ip_hdr(skb)->protocol;    int hash, raw;    const struct net_protocol *ipprot;    resubmit;    /* 将数据包传递给对应的原始套接字

    /*      raw = raw_local_deliver(skb, protocol);    /* 得到协议表的桶索引

    /*      hash = protocol & (MAX_INET_PROTOS - 1);    /* 得到注册协议

    /*      ipprot = rcu_dereference(inet_protos[hash]);    if (ipprot != NULL) {    int ret;    if (!net_eq(net, &init_net) && !ipprot->netns_ok) {

xfrm策略检查

    /*      if (!ipprot->no_policy) {    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {    kfree_skb(skb);    goto out;

    /*      ret = ipprot->handler(skb);    if (ret < 0) {    protocol = -ret;    goto resubmit;    }    IP_INC_STATS_BH(net,

    /*      if (!raw) {    if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {    /* 没有任何已注册的网络协议可以处理这个数据包，因此回复

    ICMP proto unreachable */    IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);    icmp_send(skb, ICMP_DEST_UNREACH,    ICMP
```

TCP协议在系统初始化时，会将对应的处理函数注册到`inet_protos`上，接下来进入TCP的处理函数`tcp_v4_rcv`中，代码如下：

```
int tcp_v4_rcv(struct sk_buff *skb) {    const struct iphdr *iph;    const struct tcphdr *th;    struct sock *sk;    int ret;    struct net *net = dev_net(skb->dev);    /* 如果数据

drop掉

    /*      if (skb->pkt_type != PACKET_HOST)    goto discard_it;    /* Count it even if it's bad */    TCP_INC_STATS_BH(net, TCP_MIB_INSEGS);    /* 数据包至少还有一个

TCP报文头部长度

    /*      if (!pskb_may_pull(skb, sizeof(struct tcphdr)))    goto discard_it;    /* 得到

TCP报文头部

    /*      th = tcp_hdr(skb);    /* 检查

TCP的数据偏移，至少要比头部大

    /*      if (th->doff < sizeof(struct tcphdr) / 4)    goto bad_packet;    /* 检查数据段长度

    /*      if (!pskb_may_pull(skb, th->doff * 4))    goto discard_it;    /* 计算校验和

    /*      if (!skb_csum_unnecessary(skb) && tcp_v4_checksum_init(skb))    goto bad_packet;    /* 重新得到

TCP头部。因为前面的代码可能会重新申请

    skb.

    /*      th = tcp_hdr(skb);    /* 得到
```

IP头部

```
*/   iph = ip_hdr(skb);   /* 设置
```

TCP控制块的序列号，结束序列号，确认序列号等

```
*/   TCP_SKB_CB(skb)->seq = htonl(th->seq);   TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin + skb->len - th->doff * 4);   TCP_SKB_CB(skb)->ack_seq =
```

sock结构。

这里先对已经连接的

sock进行查找，然后对监听的

sock进行查找

```
*/   sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);   if (!sk)       goto no_tcp_socket;process:   /* 如果
```

sock处于

TIME_WAIT状态，则跳转到

```
do_time_wait */   if (sk->sk_state == TCP_TIME_WAIT)       goto do_time_wait;   /* 如果数据包的
```

TTL小于设置的

TTL两值，则丢弃

```
*/   if (unlikely(iph->ttl < inet_sk(sk)->min_ttl)) {       NET_INC_STATS_BH(net, LINUX_MIB_TCPMINTTLDROP);       goto discard_and_relse;   }   /* xfrm策略失败
```

```
*/   if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))       goto discard_and_relse;   /* 虽然函数的名字为
```

reset重置，但实际上是释放了

netfilter的相关资源

```
*/   nf_reset(skb);   /* 执行
```

socket过滤器

```
*/   if (sk_filter(sk, skb))       goto discard_and_relse;   /* 重置数据包的网卡信息
```

```
*/   skb->dev = NULL;   /* 锁住
```

sock. 获得控制权

```
*/   bh_lock_sock_nested(sk);   ret = 0;   if (!sock_owned_by_user(sk)) {       /* 如果用户进程没有再使用这个
```

sock */ /* 由

DMA来做数据包拷贝，实现

```
TCP receive offload*/#ifdef CONFIG_NET_DMA   struct tcp_sock *tp = tcp_sk(sk);   if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)       tp->ucopy.dma_chan = dma_fin
```

prequeue中

```
*/           if (!tcp_prequeue(sk, skb)) {               /* 如果放到
```

prequeue中失败，则只能即时处理该数据包


```
*/          ret = tcp_v4_do_rcv(sk, skb);          }          }          /*      else if的时候，意味着用户进程正在使用这个套接字。

      那么就把数据包保存到

backlog中。

      */      } else if (unlikely(sk_add_backlog(sk, skb))) {          bh_unlock_sock(sk);          NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);          goto discard_and_relse;      }

sk的控制权

*/      sock_put(sk);      return ret;no_tcp_socket:      /* 若没有找到对应的

TCP套接字，并且

xfrm策略检测失败，则丢弃数据包

*/      if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))          goto discard_it;      /* 检查是否数据包长度出错，或者是校验和出错

*/      if (skb->len < (th->doff << 2) || tcp_checksum_complete(skb)) { bad_packet:          TCP_INC_STATS_BH(net, TCP_MIB_INERRS);      } else {          /* 若数据包未出错，但也没有匹配的套接字，则

TCP RESET */          tcp_v4_send_reset(NULL, skb);      }discard_it:      /* Discard frame. */      kfree_skb(skb);      return 0;discard_and_relse:      sock_put(sk);      goto discard_it;do_ti

TIME_WAIT状态的数据包处理。

*/      /* 在此省略了这些代码分析

*/      }
```

根据上面的分析可知，对于SYN包的处理，还需要进入函数tcp_v4_do_rcv，代码如下：

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb){      struct sock *rsk;      #ifdef CONFIG_TCP_MD5SIG          /* 进行

TCP的

MD5检查，是

TCP的一个安全检查

*/      if (tcp_v4_inbound_md5_hash(sk, skb)          goto discard;      #endif      if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */          /* 处理已连接的数据包流程

*/          /* 如果数据包与套接字的

rxhash不同，则重置套接字的

rxhash*/          sock_rps_save_rxhash(sk, skb);          /* 进入已连接

TCP的处理函数

*/          if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {              rsk = sk;              goto reset;          }          return 0;      }      /* 运行到这里，则表明该

TCP为非连接状态

*/          /* 检查数据包长度和

TCP校验值

*/      if (skb->len < tcp_hdrlen(skb) || tcp_checksum_complete(skb))          goto csum_err;      if (sk->sk_state == TCP_LISTEN) {          /* 连接处于监听状态，这是我们要跟踪的流程

*/          /*          处理连接请求，即处理
```

SYN包。作为第一个

SYN包请求，服务端只有

一个监听

sock，所以这里返回的

nsk实际上就是

sk。

这里就不跟踪

tcp_v4_hnd_req了。

```
/* struct sock *nsk = tcp_v4_hnd_req(sk, skb); if (!nsk) goto discard; if (nsk != sk) { /* 为
```

SYN请求生成了新的

sock结构，自然需要重新做

```
RFS hash */ sock_rps_save_rxhash(nsk, skb); /* 处理子
```

```
sock */ if (tcp_child_process(sk, nsk, skb)) { rsk = nsk; goto reset; } return 0; } /* 根据状态处理数据包
```

```
*/ if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) { rsk = sk; goto reset; } /* 省略其余的不相干的代码
```

```
*/}
```

进入tcp_rcv_state_process，我们截取部分相关的代码：

```
switch (sk->sk_state) { /* sock处于监听状态
```

```
*/case TCP_LISTEN: /* 监听状态不应收到
```

ack包

```
*/ if (th->ack) return 1; /* 丢弃
```

RST数据包

```
*/ if (th->rst) goto discard; /* 收到
```

SYN请求包

```
*/if (th->syn) { /* 若设置了
```

FIN结束标志，则丢弃包

```
*/ if (th->fin) goto discard; /* 调用对应的处理连接请求的回调函数
```

```
*/ if (icsk->icsk_af_ops->conn_request(sk, skb) < 0) return 1; /* SYN包处理完毕，释放数据包
```

```
*/ kfree_skb(skb); return 0; }
```

对于IPv4的TCP来说，处理连接请求的函数是tcp_v4_conn_request，代码如下：

```
int tcp_v4_conn_request(struct sock *gsk, struct sk_buff *skb){ struct tcp_extend_values tmp_ext; struct tcp_options_received tmp_opt; const u8 *hash_location; stru
```

```

*/ if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST)) goto drop; /* TW buckets are converted to open requests without * limitations, they cons

*/ if (inet_csk_reqsk_queue_is_full(sk) && !isn) { /* 判断是否使用

syn cookie. 如不使用则丢弃该包

*/ want_cookie = tcp_syn_flood_action(sk, skb, "TCP"); if (!want_cookie) goto drop; } /* backlog队列已满，并且队列中已有足够多的最近未处理的连接请求。则丢弃该包

*/ if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) goto drop; /* 申请一个请求

sock */ req = inet_reqsk_alloc(&tcp_request_sock_ops); if (!req) goto drop;#ifdef CONFIG_TCP_MD5SIG tcp_rsk(req)->af_specific = &tcp_request_sock_ipv4_ops;#endi

TCP选项

*/ tcp_clear_options(&tmp_opt); tmp_opt.mss_clamp = TCP_MSS_DEFAULT; tmp_opt.user_mss = tp->rx_opt.user_mss; tcp_parse_options(skb, &tmp_opt, &hash_location, 0);

syn cookie */ if (tmp_opt.cookie_plus > 0 && tmp_opt.saw_timestamp && !tp->rx_opt.cookie_out_never && (sysctl_tcp_cookie_size > 0 || (tp->cookie_va

syn cookie. 但没有时间戳选项，则清除

TCP选项

*/ if (want_cookie && !tmp_opt.saw_timestamp) tcp_clear_options(&tmp_opt); /* 初始化

request sock */ tmp_opt.timestamp_ok = tmp_opt.saw_timestamp; tcp_openreq_init(req, &tmp_opt, skb); ireq = inet_rsk(req); ireq->loc_addr = daddr; ireq->rmt_addr = sadd

syn cookie. 或者有时间戳选项时，如果请求表示支持

ECN. 则服务器端也设置

ECN标志

*/ if (!want_cookie || tmp_opt.timestamp_ok) TCP_ECN_create_request(req, tcp_hdr(skb)); if (want_cookie) { /* 若需要做

syn cookie. 则产生一个

cookie序号

*/ isn = cookie_v4_init_sequence(sk, skb, &req->mss); req->cookie_ts = tmp_opt.timestamp_ok; } else if (!isn) { struct inet_peer *peer = NULL; stru

TimeWait状态的

socket是否可以重用：

1) 该

socket支持时间戳选项。

2) 打开了

TimeWait状态

socket重用开关。

3) 通过查找路由，获得对端

peer信息。

4) 当前时间与对端的上个时间戳间隔小于

```

TCP_PAWS_MSL (

60) 秒, 并且新请求时间小于对端

上个时间戳

TCP_PAWS_MSL (

60) 秒以上.

当同时满足上面几个条件时, 则认为该请求为非法请求

*/ if (tmp_opt.saw_tstamp && tcp_death_row.sysctl_tw_recycle && (dst = inet_csk_route_req(sk, &fl4, req)) != NULL && fl4.daddr =

syn cookie的情况下, 内核对

syn flood做的简单防护:

1) 连接队列已经使用了四分之三以上.

2) 没有对端信息或对端没有时间戳.

3) 没有路由信息或没有路由的

RTT时间.

当同时满足以上条件时, 表明队列已接近满队列, 同时这个新连接可能无法正常通信, 那么

就会放弃这个请求

*/ else if (!sysctl_tcp_syncookies && (sysctl_max_syn_backlog - inet_csk_reqsk_queue_len(sk) < (sysctl_max_syn_backlog >> 2)) &&

/ isn = tcp_v4_init_sequence(sk); } / 保存初始序列号和

synack发送时间

/ tcp_rsk(req)->snt_isn = isn; tcp_rsk(req)->snt_synack = tcp_time_stamp; / 回复

SYNACK数据包

*/ if (tcp_v4_send_synack(sk, dst, req, (struct request_values *)&tmp_ext) || want_cookie) goto drop_and_free; /* 将

request sock加入到哈希表中

*/ inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT); return 0;drop_and_release: dst_release(dst);drop_and_free: reqsk_free(req);drop: return 0;}

这就是服务端收到SYN包, 并回复SYN+ACK的过程。

12.5.3 接收SYN+ACK数据包

客户端接收SYN+ACK数据包的流程与服务器端类似，都要经过ip_local_deliver_finish->tcp_v4_rcv->tcp_v4_do_rcv->tcp_rcv_state_process，这里会根据TCP的不同连接状态，进行不同的处理。对于此时的客户端来说，其连接状态为TCP_SYN_SENT（即发送了SYN包），其代码如下：

```
case TCP_SYN_SENT: /* 处理

SYN+ACK, 完成三次握手

*/   queued = tcp_rcv_synsent_state_process(sk, skb, th, len);   if (queued >= 0)   return queued;   /* Do step6 onward by hand. */   /* 处理

urgent数据

*/   tcp_urg(sk, skb, th);   __kfree_skb(skb);   tcp_data_snd_check(sk);
```

然后进入tcp_rcv_synsent_state_process，代码如下：

```
static int tcp_rcv_synsent_state_process(struct sock *sk, struct sk_buff *skb, const struct tcphdr *th, unsigned int len){   const u8 *hash_location;   struc

TCP选项

*/   tcp_parse_options(skb, &tp->rx_opt, &hash_location, 0);   /* 设置了

ACK标志

*/   if (th->ack) {   /* rfc793:   * "If the state is SYN-SENT then   *   first check the ACK bit   *   If the ACK bit is set   *   If SEG

ACK号非法，即不等于我们下次要发的序列号，则重置连接

*/   if (TCP_SKB_CB(skb)->ack_seq != tp->snd_nxt)   goto reset_and_undo;   /* 判断

TCP时间戳是否合法

*/   if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&   !between(tp->rx_opt.rcv_tsecr, tp->retrans_stamp,   tcp_time_stamp)) {   NET_IN

ACK标志已经通过了检查。这时，如果设置了

Reset位，则重置连接

*/   if (th->rst) {   tcp_reset(sk);   goto discard;   }   /* 如果没有设置

SYN标志，则丢弃该包

*/   if (!th->syn)   goto discard_and_undo;   /* 如果

TCP套接字设置了

ECN标志，但是数据包没有设置

ECE标志（表示对端不支持

TCP ECN显示拥塞通告），则清除掉本端的

ECN标志）

*/   TCP_ECN_rcv_synack(tp, th);   /* 处理

ack数据包，设置

TCP发送窗口
```

```
*/      tp->snd_wll = TCP_SKB_CB(skb)->seq;      tcp_ack(sk, skb, FLAG_SLOWPATH);      /* Ok.. it's good. Set up sequence numbers and      * move to established.
```

TCP窗口大小, 是不考虑

scale选项的

```
*/      tp->snd_wnd = ntohs(th->window);      tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);      /* 如果没有
```

windows scale选项

```
*/      if (!tp->rx_opt.wscale_ok) {      /* 将接收端和接收端的窗口扩展选项设置为
```

```
0 */      tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;      /* 设置窗口的最大值
```

```
*/      tp->window_clamp = min(tp->window_clamp, 65535U);      }      /* 判断是否时间戳选项
```

```
*/      if (tp->rx_opt.saw_tstamp) {      /* 设置时间戳选项
```

```
*/      tp->rx_opt.tstamp_ok      = 1;      tp->tcp_header_len =      sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;      tp->advms      -= TC
```

```
*/      tcp_mtup_init(sk);      tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);      tcp_initialize_rcv_mss(sk);      /* Remember, tcp_poll() does not lock socket!      *
```

```
*/      tp->copied_seq = tp->rcv_nxt;      if (cvp != NULL &&      cvp->cookie_pair_size > 0 &&      tp->rx_opt.cookie_plus > 0) {      int cookie_size =
```

```
*/      tcp_set_state(sk, TCP_ESTABLISHED);      security_inet_conn_established(sk, skb);      /* Make sure socket is routed, for correct metrics. */      /* 查找路由
```

```
*/      icsk->icsk_af_ops->rebuild_header(sk);      /* 下面对路由的
```

metric.

TCP的阻塞控制和缓存等进行初始化

```
*/      tcp_init_metrics(sk);      tcp_init_congestion_control(sk);      /* Prevent spurious tcp_cwnd_restart() on first data      * packet.      */      tp->lsn
```

keepalive. 则初始化

keepalive定时器

```
*/      if (sock_flag(sk, SOCK_KEEPOPEN)      inet_csk_reset_keepalive_timer(sk, keepalive_time_when(tp));      /* 若发送方没有设置窗口扩展选项, 则设置
```

TCP快速路径预测标志

```
*/      if (!tp->rx_opt.snd_wscale)      __tcp_fast_path_on(tp, tp->snd_wnd);      else      tp->pred_flags = 0;      /* 如果
```

sock的状态不是死亡状态

```
*/      if (!sock_flag(sk, SOCK_DEAD)) {      /* 改变
```

sock状态, 唤醒等待进程

```
*/      sk->sk_state_change(sk);      /* 若有异步等待队列, 则给该进程发送异步事件
```

```
*/      sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);      }      /*      如果该套接字:
```

1) 有写操作等待.

2) 设置了延迟

accept.

3) 没有设置快速

ack.

```
/*      if (sk->sk_write_pending ||      icsk->icsk_accept_queue.rskq_defer_accept ||      icsk->icsk_ack.pingpong) {      /* 满足上面条件之一，则延时确认

/*      /* Save one ACK. Data will be ready after      * several ticks, if write_pending is set.      *      * It may be deleted, but with this feature

/*      tcp_send_ack(sk);      }      return -1;} /* 到此，表示没有
```

ACK标志

```
/*      if (th->rst) {      /* 如果没有
```

ACK只有

RST，则丢弃该包

```
/*      goto discard_and_undo;      } /* 时间戳检测

/*      if (tp->rx_opt.ts_recent_stamp && tp->rx_opt.saw_tstamp &&      tcp_paws_reject(&tp->rx_opt, 0))      goto discard_and_undo;      if (th->syn) {      /*      若只有
```

SYN标志，没有

ACK，则可能是同时发出了多个

SYN连接请求，甚至有可能是自己连接自己

```
/*      /* 设置连接状态为收到
```

SYN包

```
/*      tcp_set_state(sk, TCP_SYN_RECV);      /*      后面的代码，与之前收到
```

SYN包的流程基本一致，在此就不做分析了

```
/*      if (tp->rx_opt.saw_tstamp) {      tp->rx_opt.tstamp_ok = 1;      tcp_store_ts_recent(tp);      tp->tcp_header_len =      sizeof(st
```

SYN也没有

RST标志，则丢弃数据包后返回

```
*/discard_and_undo:      /* 丢弃数据包
```

```
/*      tcp_clear_options(&tp->rx_opt);      tp->rx_opt.mss_clamp = saved_clamp;      goto discard;reset_and_undo:      /* 重置连接。与丢弃数据包的区别在于返回值。非
```

0时，调用者会重置连接

```
/*      tcp_clear_options(&tp->rx_opt);      tp->rx_opt.mss_clamp = saved_clamp;      return 1;}
```

12.5.4 接收ACK数据包，完成三次握手

在前文中，我们已经知道了发往本机的TCP数据包会进入tcp_v4_do_rcv。但因为此时还未真正地完成三次握手，所以TCP仍然是未连接状态，自然就会再次进入函数tcp_v4_hnd_req了。

```
static struct sock *tcp_v4_hnd_req(struct sock *sk, struct sk_buff *skb){
    struct tcphdr *th = tcp_hdr(skb);
    const struct iphdr *iph = ip_hdr(skb);
    struct sock *nsk;

    SYN请求时，已经将对应的

    request_sock加入到了队列中，因此这次收到

    ACK答复时，是

    可以找到对应的

    request_sock的。

    另外需要注意的是，这个函数还会有一个输出值

    prev. 其为返回值

    req前面的元素。之所以返回这个

    prev值，是为了在后面的

    tcp_check_req函数中，移除

    req时，不需要进行第二次查找

    /*
     * struct request_sock *req = inet_csk_search_req(sk, &prev, th->source,
     * iph->saddr, iph->daddr);
     * if (req)
     *     retur

    /*
     * ____

}


```

下面进入tcp_check_req函数：

```
struct sock *tcp_check_req(struct sock *sk, struct sk_buff *skb,
                           struct request_sock *req,
                           struct request_sock **prev){
    struct tcp_options_receiv

    saw_tstamp. 因为时间戳选项依赖于每个数据包

    /*
     * tmp_opt.saw_tstamp = 0;
     * if (th->doff > (sizeof(struct tcphdr)>>2)) {
     *     /* 若实际数据位置偏移量大于

    TCP固定报头长度，则表明该报文一定包含了

    TCP选项

    /*
     *     /* 解析

    TCP选项

    /*
     *     tcp_parse_options(skb, &tmp_opt, &hash_location, 0);
     *     /* 判断是否有时间戳选项

    /*
     *     if (tmp_opt.saw_tstamp) {
     *         /* 检查时间戳选项

    /*
     *         tmp_opt.ts_recent = req->ts_recent;
     *         /* We do not store true stamp, but it is not required,
     *         * it can be estimated (approximately)

    SYN包为重传的

    SYN包，回复

```



```
SYN+ACK*/      req->rsk_ops->rtx_syn_ack(sk, req, NULL);      return NULL;      }      /* 非法的
```

ACK值

```
*/      if ((flg & TCP_FLAG_ACK) &&      (TCP_SKB_CB(skb)->ack_seq !=      tcp_rsk(req)->snt_isn + 1 + tcp_s_data_size(tcp_sk(sk))))      return sk;      /* 时间戳检查失败, 或
```

```
*/      if (paws_reject || !tcp_in_window(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,      tcp_rsk(req)->rcv_isn + 1, tcp_rsk(req)->rcv_isn + 1 + req->rcv_wnd)) {
```

ACK确认

```
*/      if (!(flg & TCP_FLAG_RST))      req->rsk_ops->send_ack(sk, skb, req);      /* 若时间戳检测失败, 则增加相应的计数
```

```
*/      if (paws_reject)      NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSESTABREJECTED);      return NULL;      }      /* In sequence, PAWS is OK. */      /* 若数据包为有序数据
```

```
*/      if (tmp_opt.saw_tstamp && !after(TCP_SKB_CB(skb)->seq, tcp_rsk(req)->rcv_isn + 1))      req->ts_recent = tmp_opt.rcv_tsval;      /* 若序列号在接收窗口之外, 则去掉
```

SYN标志

```
*/      if (TCP_SKB_CB(skb)->seq == tcp_rsk(req)->rcv_isn) {      /* Truncate SYN, it is out of window starting      at tcp_rsk(req)->rcv_isn + 1. */      flg &= ~TCP_F
```

SYN和

RST标志, 若都设置了, 则将当前半连接从队列中清除

```
*/      if (flg & (TCP_FLAG_RST|TCP_FLAG_SYN)) {      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);      goto embryonic_reset;      }      /* 若没有设置
```

ACK, 则丢弃该包

```
*/      if (!(flg & TCP_FLAG_ACK))      return NULL;      /* While TCP_DEFER_ACCEPT is active, drop bare ACK. */      /* 如果设置了延迟接收, 则丢弃单独的
```

ACK包

```
*/      if (req->retrans < inet_csk(sk)->icsk_accept_queue.rskq_defer_accept &&      TCP_SKB_CB(skb)->end_seq == tcp_rsk(req)->rcv_isn + 1) {      inet_rsk(req)->acked = 1;
```

SYN+ACK时间

```
*/      if (tmp_opt.saw_tstamp && tmp_opt.rcv_tsecr)      tcp_rsk(req)->snt_synack = tmp_opt.rcv_tsecr;      else if (req->retrans) /* don't take RTT sample if retrans && ~TS
```

TCP的三次握手已经完成, 使用

syn_rcv_sock创建真正的套接字

```
*/      child = inet_csk(sk)->icsk_af_ops->syn_rcv_sock(sk, skb, req, NULL);      if (child == NULL)      goto listen_overflow;      /* 新的套接字已经创建, 因此原来的
```

request sock可以从队列中删除

```
*/      inet_csk_reqsk_queue_unlink(sk, req, prev);      inet_csk_reqsk_queue_removed(sk, req);      /* 将套接字加入到已连接的队列中
```

```
*/      inet_csk_reqsk_queue_add(sk, req, child);      return child;listen_overflow:      if (!sysctl_tcp_abort_on_overflow) {      inet_rsk(req)->acked = 1;      return NULL;
```

这样, 当tcp_check_req成功返回时, 会返回一个新创建的sock结构。那么在tcp_v4_do_rcv中, 就会进入tcp_child_process中。

```
int tcp_child_process(struct sock *parent, struct sock *child,      struct sk_buff *skb){      int ret = 0;      int state = child->sk_state;      /* 检查
```

sock是否正在被用户进程使用

```
*/      if (!sock_owned_by_user(child)) {      /* 用户进程没有占用
```

sock的情况

```
*/      ret = tcp_rcv_state_process(child, skb, tcp_hdr(skb),          skb->len);          /* Wakeup parent, send SIGIO */          /* 唤醒阻塞在父

sock的任务

*/      if (state == TCP_SYN_RECV && child->sk_state != state)          parent->sk_data_ready(parent, 0);          } else {          /* 由于用户进程占用着

sock. 将数据包加入

backlog. 以后再处理

*/      __sk_add_backlog(child, skb);          }      bh_unlock_sock(child);      sock_put(child);      return ret;}
```

这里我们考虑数据包被立刻处理的情况，即用户进程没有占用sock结构，那么这里数据还是会进入tcp_rcv_state_process的。根据前面的分析，tcp_rcv_state_process是根据套接字的状态来处理数据包的。而child是从父sock生成的，所以如果child的状态和父sock的状态一致，肯定是有问题的——因为父sock的状态是监听状态。那么child的状态是何时改变的呢？

让我们退回到创建child的函数tcp_v4_syn_recv_sock->tcp_create_openreq_child->inet_csk_clone中，代码如下：

```
struct sock *inet_csk_clone(struct sock *sk, const struct request_sock *req,          const gfp_t priority){          /* 克隆一个新的

sock结构

*/      struct sock *newsk = sk_clone(sk, priority);      if (newsk != NULL) {          /* 开始克隆面向连接的

sock信息

*/          struct inet_connection_sock *newicsk = inet_csk(newsk);          /* 将新

sock设置为

TCP_SYN_RECV状态

*/          newsk->sk_state = TCP_SYN_RECV;          newicsk->icsk_bind_hash = NULL;          /* 后面是复制其他变量的代码，在此省略掉

*/          }      return newsk;}
```

这个函数的命名稍稍有些别扭，名字叫做clone（克隆），也就是说，所有的内容都应该保持一致。而这里在这个inet_csk_clone后，新sock的状态与父sock的状态并不一致。

下面来查看tcp_rcv_state_process处理TCP_SYN_RECV状态的代码：

```
case TCP_SYN_RECV:          /* acceptable是

tcp_rcv_state_process在前面对

ACK数据包进行的判断。

*/      if (acceptable) {          /* 这时已经完成了三次握手

*/          /* 初始化用户态未读数据的序列号是我们期待接收的下一个序列号

*/          tp->copied_seq = tp->rcv_nxt;          smp_mb();          /* 设置连接状态为已连接

*/          tcp_set_state(sk, TCP_ESTABLISHED);          /*          sk_state_change为一个回调函数，默认为

sock_def_wakeup, 其会唤醒

sleep在该
```

socket的进程

```
*/          sk->sk_state_change(sk);          /* 若该
```

sock有对应的用户态

socket，则执行异步

I/O通知

```
*/          /*          这里需要注意的是，对于我们目前的情况来说，子
```

sock是从监听

sock clone而来的，其中

```
sk_sleep和
```

sk_socket都是

NULL。

那么三次握手以后，阻塞在监听

socket的进程是如何被唤醒的呢？

```
tcp_child_process在调用
```

tcp_rcv_state_process后，会检查

sock状态是否发生了变

化，如果发生了变化，则会调用

parent->sk_data_ready(parent, 0);这样，就可以将事

件通知到阻塞在监听

sock的进程了。

```
*/          if (sk->sk_socket)          sk_wake_async(sk,          SOCK_WAKE_IO, POLL_OUT);          /* 初始化未确认回复的序列号
```

```
*/          tp->snd_una = TCP_SKB_CB(skb)->ack_seq;          /* 初始化发送窗口
```

```
*/          tp->snd_wnd = ntohs(th->window) <<          tp->rx_opt.snd_wscale;          tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);          /* 如果有时间戳选项，则
```

MSS需要减去时间戳所占的大小

```
*/          if (tp->rx_opt.timestamp_ok)          tp->advms = TCPOLEN_TSTAMP_ALIGNED;          /* Make sure socket is routed, for          * correct metrics.          */          ic
```

```
*/          tcp_init_metrics(sk);          tcp_init_congestion_control(sk)          /* Prevent spurious tcp_cwnd_restart() on          * first data packet.          */          tp->lsnd
```

目前为止，三次握手的源码分析已经结束了。其内部还有很多细节值得展开学习，但那就不是一两个章节所能完成的任务了。笔者只是抛砖引玉，给出一个脉络，关于剩下的细节大家可以自己通过阅读代码来完善。

第13章 网络通信：数据报文的发送

第12章学习了Linux套接字的创建、监听和连接，并重点分析了TCP建立连接时的三次握手过程。本章将从应用层到内核来研究数据包的发送过程。

13.1 发送相关接口

Linux内核为套接字提供了多个发送数据的接口，接口定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

`send`只能用于处理已连接状态的套接字（注意，从第11章的内容已经知道，无论是UDP还是TCP，都可以进行连接）。而`sendto`可以在调用时，指定目的地址。这样的话，如果套接字已经是连接状态，那么目的地址`dest_addr`与地址长度就应该为NULL和0，不然就可能会返回错误。`sendmsg`则比较特殊，无论是要发送的数据还是目的地址，都保存在`msg`中。其中`msg.msg_name`和`msg.msg_len`用于指明目的地址，而`msg.msg_iov`则用于保存要发送的数据。这三个系统调用都支持设置指示标志位`flags`。



说明

稍微现代些的系统调用，一般都会拥有或保留一个指示标志参数。通过标志位`flags`，可以从容地为系统调用增加新功能，并同时兼容老版本。第1章中介绍的`dup`、`dup2`和`dup3`则是这方面的一个反面典型。在不支持`flag`的情况下，不得不一再创建新的`dup`接口，直到`dup3`加入了对`flag`的支持为止。

由于`socket`同时还是文件描述符，所以为文件提供的写操作（如`write`、`writv`等），也可以被`socket`套接字直接调用，在此就不重复叙述了。

13.2 数据包从用户空间到内核空间的流程

从13.1节可知，`socket`套接字在发送数据包时有多个系统调用，既有套接字本身的发送接口，又可以重用文件描述符的写操作。这些不同的接口是否会导致数据包从用户空间发送到内核空间时走向不同的流程呢？下面让我们通过阅读源码来回答这个问题。

`send`的内核实现代码如下：

```
SYSCALL_DEFINE4(send, int, fd, void __user *, buff, size_t, len,
                 unsigned, flags)
{
    /*
     send可以视为

sendto的一种特例，即不设置目的地址的

sendto调用。

所以内核实现也是让

send直接调用

sendto。

*/
    return sys_sendto(fd, buff, len, flags, NULL, 0);
}
```

既然其内核实现是让`send`直接调用`sendto`，那么，下面我们就来看一下`sendto`的内核实现，代码如下：

```
SYSCALL_DEFINE6(sendto, int, fd, void __user *, buff, size_t, len,
                 unsigned, flags, struct sockaddr __user *, addr,
                 int, addr_len)
{
    struct socket *sock;
    struct sockaddr_storage address;
    int err;
    struct msghdr msg;
    struct iovec iov;
    int fput_needed;
    /* 长度合法性检查

*/
    if (len > INT_MAX)
        len = INT_MAX;
    /* 从文件描述符获得套接字
```

socket的结构

```
*/
sock = sockfd_lookup_light(fd, &err, &fput_needed);
if (!sock)
    goto out;
/* 将数据转换为
```

iovec结构，来调用后面的

```
sendmsg */
iov.iov_base = buff;
iov.iov_len = len;
msg.msg_name = NULL;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
msg.msg_control = NULL;
msg.msg_controllen = 0;
msg.msg_namelen = 0;
/* 如果设置了地址，则设置
```

```
msg_name */if (addr) {      /* 将地址参数复制到内核变量中
```

```
*/
err = move_addr_to_kernel(addr, addr_len, (struct sockaddr *)&address);
if (err < 0)
    goto out_put;
msg.msg_name = (struct sockaddr *)&address;
msg.msg_namelen = addr_len;
}
/*如果
```

socket设置了非阻塞，则消息的标志设置为

DONTWAIT（其实也是非阻塞的语义）

```
*/
if (sock->file->f_flags & O_NONBLOCK)
    flags |= MSG_DONTWAIT;
msg.msg_flags = flags;
/* 调用
```

sock_sendmsg来发送数据包

```
*/
err = sock_sendmsg(sock, &msg, len);
out_put:
fput_light(sock->file, fput_needed);
out:
return err;
}
```

这里又调用到sock_sendmsg了，从名字上就能感觉到它可能也会被第三个接口sendmsg所调用。下面让我们来验证这个猜想。

```

SYSCALL_DEFINE3(sendmsg, int, fd, struct msghdr __user *, msg, unsigned, flags)
{
    int fput_needed, err;
    struct msghdr msg_sys;
    /* 通过文件描述符获得

```

socket套接字结构

```

*/
    struct socket *sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (!sock)
        goto out;
    /* 调用

```

__sys_sendmsg来发送数据包

```

*/
    err = __sys_sendmsg(sock, msg, &msg_sys, flags, NULL);
    fput_light(sock->file, fput_needed);
out:
    return err;
}

```

接下来进入__sys_sendmsg, 代码如下:

```

static int __sys_sendmsg(struct socket *sock, struct msghdr __user *msg,
                        struct msghdr *msg_sys, unsigned flags,
                        struct ucred *ucred)
{
    struct compat_msghdr __user *msg_compat =
        (struct compat_msghdr __user *)msg;
    struct sockaddr_storage address;
    struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
    unsigned char ctl[sizeof(struct cmsghdr) + 20]
        __attribute__((aligned(sizeof(__kernel_size_t)))));
    /* 20 is size of ipv6_pktinfo */
    unsigned char *ctl_buf = ctl;
    int err, ctl_len, iov_size, total_len;
    err = -EFAULT;
    /* 从用户空间得到用户消息

*/
    if (MSG_COMPAT & flags) {
        /* 紧凑消息类型

*/
        if (get_compat_msghdr(msg_sys, msg_compat))
            return -EFAULT;
        } else if (copy_from_user(msg_sys, msg, sizeof(struct msghdr)))
            return -EFAULT;
        /* do not move before msg_sys is valid */
        err = -EMSGSIZE;
        /* 消息数据块个数检查

*/
        if (msg_sys->msg_iovlen > UIO_MAXIOV)
            goto out;
        /* Check whether to allocate the iovec area */
        err = -ENOMEM;
        /* 在内核空间申请消息数据长度

*/
        iov_size = msg_sys->msg_iovlen * sizeof(struct iovec);
        if (msg_sys->msg_iovlen > UIO_FASTIOV) {

```



```

        iov = sock_kmalloc(sock->sk, iov_size, GFP_KERNEL);
        if (!iov)
            goto out;
    }
    /* This will also move the address data into kernel space */
    /* 前面只是将消息头, 或者说消息的结构体, 复制到内核空间, 现在是将消息的真正内容, 即

```

iov的内容复制到内核空间

```

*/
    if (MSG_CMSG_COMPAT & flags) {
        err = verify_compat_iovec(msg_sys, iov,
                                   (struct sockaddr *)&address,
                                   VERIFY_READ);
    } else
        err = verify_iovec(msg_sys, iov,
                            (struct sockaddr *)&address,
                            VERIFY_READ);
    if (err < 0)
        goto out_freeiov;
    total_len = err; err = -ENOBUFFS; /* 与消息数据块类似, 复制控制消息块, 就不详细描述了

*/
    if (msg_sys->msg_controllen > INT_MAX)
        goto out_freeiov;
    ctl_len = msg_sys->msg_controllen;
    if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
        err =
            cmsghdr_from_user_compat_to_kern(msg_sys, sock->sk, ctl,
                                              sizeof(ctl));
        if (err)
            goto out_freeiov;
        ctl_buf = msg_sys->msg_control;
        ctl_len = msg_sys->msg_controllen;
    } else if (ctl_len) {
        if (ctl_len > sizeof(ctl)) {
            ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
            if (!ctl_buf)
                goto out_freeiov;
        }
        err = -EFAULT;
        /*
         * Careful! Before this, msg_sys->msg_control contains a user pointer.
         * Afterwards, it will be a kernel pointer. Thus the compiler-assisted
         * checking falls down on this.
         */
        if (copy_from_user(ctl_buf,
                           (void __user __force *)msg_sys->msg_control,
                           ctl_len))
            goto out_freectl;
        msg_sys->msg_control = ctl_buf; /* 设置消息标志

*/
    msg_sys->msg_flags = flags;
    /* 如果套接字是非阻塞的, 则设置消息标志

MSG_DONTWAIT */
    if (sock->file->f_flags & O_NONBLOCK)
        msg_sys->msg_flags |= MSG_DONTWAIT;
    /* 如果这次发送的目的地址与上次成功发送的目的地址一致, 那就可以省略安全性检查

*/
    if (used_address && msg_sys->msg_name &&
        used_address->name_len == msg_sys->msg_namelen &&
        !memcmp(&used_address->name, msg_sys->msg_name,
                used_address->name_len)) {
        /* 调用不进行安全性检查的函数

*/
    err = sock_sendmsg_nosec(sock, msg_sys, total_len);
    goto out_freectl;
}

```

```
/* 调用
```

sock_sendmsg, 需要安全性检查, 最终仍然会调用到

sock_send_msg_nosec函数

```
*/
err = sock_sendmsg(sock, msg_sys, total_len);
/* 如果本次发送成功, 则保存当前的目的地址

*/
if (used_address && err >= 0) {
    used_address->name_len = msg_sys->msg_namelen;
    if (msg_sys->msg_name)
        memcpy(&used_address->name, msg_sys->msg_name,
               used_address->name_len);
}
out_freectl:
if (ctl_buf != ctl)
    sock_kfree_s(sock->sk, ctl_buf, ctl_len);
out_freeiov:
if (iov != iovstack)
    sock_kfree_s(sock->sk, iov, iov_size);
out:
    return err;
}
```

看完了__sys_sendmsg, 我们可以确定, 无论是哪个发送数据的系统调用, 最终都会调用到sock_sendmsg。下面是sock_sendmsg的相关代码:

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
{
    /* kiocb为内核通用的
```

IO请求结构

```
*/
struct kiocb iocb;
struct sock_iocb siocb;
int ret;
/* 初始化同步的内核
```

IO请求结构

```
*/
init_sync_kiocb(&iocb, NULL);
iocb.private = &siocb;
/* 发送消息

*/
ret = __sock_sendmsg(&iocb, sock, msg, size);
/* 返回结果表明该消息已经加入队列, 要等待完成事件

*/
if (-EIOCBQUEUED == ret)
    ret = wait_on_sync_kiocb(&iocb);
```

```
    return ret;
}
```

这里__sock_sendmsg只是做了安全性检查，然后就调用了__sock_sendmsg_nosec函数。再继续看__sock_sendmsg_nosec，代码如下：

```
static inline int __sock_sendmsg_nosec(struct kiocb *iocb, struct socket *sock,
                                     struct msghdr *msg, size_t size)
```

```
{
    /* 获得套接字在
```

```
sock_sendmsg中设置的
```

```
IO请求，
```

```
*/
    struct sock_iocb *si = kiocb_to_siocb(iocb);
    sock_update_classid(sock->sk);
    /* 初始化套接字的
```

```
IO请求字段
```

```
*/
    si->sock = sock;
    si->scm = NULL;
    si->msg = msg;
    si->size = size;
    /* 根据不同的套接字类型，调用其发送数据函数
```

```
*/
    return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

到此，我们完成了数据包从用户空间到内核空间的流程跟踪。接下来的数据包发送过程，将根据不同的协议，走不同的流程。

13.3 UDP数据包的发送流程

前文已经跟踪了数据包从用户空间到内核空间的流程，本节将以比较简单的UDP协议为例，继续跟踪数据包的发送流程——因为UDP是无连接状态的协议，所以不会给我们的代码分析带来额外的麻烦。

UDP的sendmsg操作函数为udp_sendmsg，代码如下：

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t len)
{
    /* 从

inet通用套接字得到

inet套接字

*/
    struct inet_sock *inet = inet_sk(sk);
    /* 从

inet通用套接字得到

UDP套接字

*/
    struct udp_sock *up = udp_sk(sk);
    struct flowi4 fl4_stack;
    struct flowi4 *fl4;
    int ulen = len;
    struct ipcm_cookie ipc;
    struct rtable *rt = NULL;
    int free = 0;
    int connected = 0;
    __be32 daddr, faddr, saddr;
    __be16 dport;
    u8 tos;
    int err, is_udplite = IS_UDPLITE(sk);
    /* 是否有数据包聚合：或者

UDP套接字设置了聚合选项，或者数据包消息指明了还有更多数据

*/
    int corkreq = up->corkflag || msg->msg_flags & MSG_MORE;
    int (*getfrag)(void *, char *, int, int, int, struct sk_buff *);
    struct sk_buff *skb;
    struct ip_options_data opt_copy;
    /* 数据包长度检查

*/
    if (len > 0xFFFF)
        return -EMSGSIZE;
    /* 检查消息标志。

UDP不支持带外数据

*/
    if (msg->msg_flags & MSG_OOB) /* Mirror BSD error message compatibility */
        return -EOPNOTSUPP;
    ipc.opt = NULL;
    ipc.tx_flags = 0;
    /* 设置正确的分片函数

*/
    getfrag = is_udplite ? udplite_getfrag : ip_generic_getfrag;
    fl4 = &inet->cork.fl.u.ip4;
    if (up->pending) {
        /* 该

UDP套接字还有待发的数据包

*/
        lock_sock(sk);
        /* 常见的上锁双重检查机制

*/
        if (likely(up->pending)) {
            /* 若待发的数据包不是

INET数据，则报错返回

*/
            if (unlikely(up->pending != AF_INET)) {
                release_sock(sk);
                return -EINVAL;
            }
            /* 调回追加数据处
```

```

*/
    goto do_append_data;
}
release_sock(sk);
}
ulen += sizeof(struct udphdr);
if (msg->msg_name) {
    /* 若指定了目标地址，则对其进行校验

*/
    struct sockaddr_in * usin = (struct sockaddr_in *)msg->msg_name;
    /* 检查长度

*/
    if (msg->msg_namelen < sizeof(*usin))
        return -EINVAL;
    /* 检查协议族。目前只支持

```

AF_INET和

AF_UNSPEC协议族

```

*/
    if (usin->sin_family != AF_INET) {
        if (usin->sin_family != AF_UNSPEC)
            return -EAFNOSUPPORT;
    }
    /* 若通过了检查，则设置目的地址与目的端口

*/
    daddr = usin->sin_addr.s_addr;
    dport = usin->sin_port;
    /* 目的端口不能为

0 */
    if (dport == 0)
        return -EINVAL;
} else {
    /* 如果没有指定目的地址和目的端口，则当前套接字的状态必须是已连接，即已经调用过

```

connect设置了目的地址

```

*/
    if (sk->sk_state != TCP_ESTABLISHED)
        return -EDESTADDRREQ;
    /* 使用之前设置的目的地址和目的端口

*/
    daddr = inet->inet_daddr;
    dport = inet->inet_dport;
    /* Open fast path for connected socket.
       Route will not be used, if at least one option is set.
    */
    connected = 1;
}
ipc.addr = inet->inet_saddr;
ipc.oif = sk->sk_bound_dev_if;
/* 设置时间戳标志

*/
err = sock_tx_timestamp(sk, &ipc.tx_flags);
if (err)
    return err;
/* 发送的消息包含控制数据

*/
if (msg->msg_controllen) {
    /* 虽然这个函数的名字叫作

```

send. 其实并没有任何发送动作，而只是将控制消息设置到

ipc中

```

*/
    err = ip_cmsg_send(sock_net(sk), msg, &ipc);
    if (err)
        return err;
    /* 设置释放

```

ipc.opt的标志

```

*/
    if (ipc.opt)
        free = 1;
    connected = 0;
}
if (!ipc.opt) {
    /* 如果没有使用控制消息指定

```

IP选项，则检查套接字的

IP选项设置。如果有，则使用套接字的

IP选项

```
*/
    struct ip_options_rcu *inet_opt;
    rcu_read_lock();
    inet_opt = rcu_dereference(inet->inet_opt);
    if (inet_opt) {
        memcpy(&opt_copy, inet_opt,
              sizeof(*inet_opt) + inet_opt->opt.optlen);
        ipc.opt = &opt_copy.opt;
    }
    rcu_read_unlock();
    saddr = ipc.addr;
    ipc.addr = faddr = daddr;
    /* 设置了严格路由
```

```
*/
    if (ipc.opt && ipc.opt->opt.srr) {
        if (!daddr)
            return -EINVAL;
        faddr = ipc.opt->opt.faddr;
        connected = 0;
    }
    tos = RT_TOS(inet->tos);
    /*
    若有下列情况之一的：
```

1) 套接字设置了本地路由标志。

2) 发送消息时，指明了不做路由。

3) 设置了

IP严格路由选项。

则设置不查找路由标志

```
*/
    if (sock_flag(sk, SOCK_LOCALROUTE) ||
        (msg->msg_flags & MSG_DONTROUTE) ||
        (ipc.opt && ipc.opt->opt.is_strictroute)) {
        tos |= RTO_ONLINK;
        connected = 0;
    }
    /* 如果目的地址是多播地址
```

```
*/
    if (ipv4_is_multicast(daddr)) {
        /* 若未指定出口接口，则使用套接字的多播接口索引
```

```
*/
    if (!ipc.oif)
        ipc.oif = inet->mc_index;
    /* 若源地址为
```

0. 则使用套接字的多播地址

```
*/
    if (!saddr)
        saddr = inet->mc_addr;
    connected = 0;
}
/* 连接标志为真，即此次发送的数据包与上次的地址相同，则判断保存的路由缓存是否还可用。
```

```
*/
    if (connected) {
        /* 从套接字检查并获得保存的路由缓存
```

```
*/
    rt = (struct rtable *)sk_dst_check(sk, 0);
}
/* 若目前路由缓存为空，则需要查找路由
```

```
*/
    if (rt == NULL) {
        struct net *net = sock_net(sk);
        fl4 = &fl4_stack;
        /* 根据套接字和数据包的信息，初始化
```

flowi4-这是查找路由的

```
key */
    flowi4_init_output(fl4, ipc.oif, sk->sk_mark, tos,
        RT_SCOPE_UNIVERSE, sk->sk_protocol,
        inet_sk_flowi_flags(sk)|FLOWI_FLAG_CAN_SLEEP,
        faddr, saddr, dport, inet->inet_sport);
```

```

security_sk_classify_flow(sk, flowi4_to_flowi(fl4));
/* 查找出口路由

*/

rt = ip_route_output_flow(net, fl4, sk);
if (IS_ERR(rt)) {
    /* 查找路由失败

*/

    err = PTR_ERR(rt);
    rt = NULL;
    if (err == -ENETUNREACH)
        IP_INC_STATS_BH(net, IPSTATS_MIB_OUTNOROUTES);
    goto out;
}
err = -EACCES;
/* 若路由是广播路由，并且套接字非广播套接字

*/

if ((rt->rt_flags & RTCF_BROADCAST) &&
    !sock_flag(sk, SOCK_BROADCAST))
    goto out;
if (connected) {
    /* 若该

```

UDP为已连接状态，则保存这个路由缓存

```

*/
    sk_dst_set(sk, dst_clone(&rt->dst));
}
/* 如果数据包设置了

```

MSG_CONFIRM标志，则是要告诉链路层，对端是可达的。调到

do_confirm处，可

以发现其实现方法是在有

neighbour信息的情况下，直接更新

neighbour确认时间戳为当前时间。

```

*/
if (msg->msg_flags & MSG_CONFIRM)
    goto do_confirm;
back_from_confirm:
saddr = fl4->saddr;
if (!ipc.addr)
    daddr = ipc.addr = fl4->daddr;
/* 没有使用

```

cork选项或

MSG_MORE标志。这也是最常见的情况。

```

*/
if (!corkreq) {
    /* 每次都生成一个

```

UDP数据包

```

*/
skb = ip_make_skb(sk, fl4, getfrag, msg->msg_iov, ulen,
    sizeof(struct udphdr), &ipc, &rt,
    msg->msg_flags);
err = PTR_ERR(skb);
/* 成功生成了数据包

```

```

*/
if (skb && !IS_ERR(skb)) {
    /* 发送

```

UDP数据包

```

*/
    err = udp_send_skb(skb, fl4);
}
goto out;
}
lock_sock(sk);
if (unlikely(up->pending)) {
    /*
    现在马上要做

```

cork处理，但发现套接字已经

cork了。

因此这是一个应用程序

bug，释放套接字锁，并返回错误。

```
    /*
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2\n");
    err = -EINVAL;
    goto out;
}
/*
 * Now cork the socket to pend data.
 */
/* 设置
```

cork中的流信息

```
    /*
    fl4 = &inet->cork.fl.u.ip4;
    fl4->daddr = daddr;
    fl4->saddr = saddr;
    fl4->fl4_dport = dport;
    fl4->fl4_sport = inet->inet_sport;
    up->pending = AF_INET;
do_append_data:
    /* 增加
```

UDP数据长度

```
    /*
    up->len += ulen;
    /* 向
```

IP数据包中追加新的数据

```
    /*
    err = ip_append_data(sk, fl4, getfrag, msg->msg_iov, ulen,
        sizeof(struct udphdr), &ipc, &rt,
        corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
    if (err) // 若发生错误，则丢弃所有未决的数据包

    udp_flush_pending_frames(sk);
    else if (!corkreq) // 若不在
```

cork即阻塞，则发送所有未决的数据包

```
    err = udp_push_pending_frames(sk);
    else if (unlikely(skb_queue_empty(&sk->sk_write_queue))) {
    /* 若没有未决的数据包，则重置未决标志
```

```
    /*
    up->pending = 0;
    }
    release_sock(sk);
out:
    /* 清理工作，释放各种资源，并增加相应的统计计数
```

```
    /*
    ip_rt_put(rt);
    if (free)
        kfree(ipc.opt);
    if (!err)
        return len;
    /*
    * ENOBUFS = no kernel mem, SOCK_NOSPACE = no sndbuf space. Reporting
    * ENOBUFS might not be good (it's not tunable per se), but otherwise
    * we don't have a good statistic (IpOutDiscards but it can be too many
    * things). We could add another new stat but at least for now that
    * seems like overkill.
    */
    if (err == -ENOBUFS || test_bit(SOCK_NOSPACE, &sk->sk_socket->flags)) {
        UDP_INC_STATS_USER(sock_net(sk),
            UDP_MIB_SNDBUFERRORS, is_udplite);
    }
    return err;
do_confirm:
    dst_confirm(&rt->dst);
    if (! (msg->msg_flags&MSG_PROBE) || len)
        goto back_from_confirm;
    err = 0;
    goto out;
}
```

一般情况下，在使用UDP发送数据包时很少会使用CORK或MSG_MORE标志，因为我们希望在每次调用发送接口时，就发送一次UDP数据包。因此可以不必考虑CORK和MSG_MORE的情况，而继续追踪udp_s

```
static int udp_send_skb(struct sk_buff *skb, struct flowi4 *fl4)
{
    struct sock *sk = skb->sk;
    struct inet_sock *inet = inet_sk(sk);
    struct udphdr *uh;
    int err = 0;
    int is_udplite = IS_UDPLITE(sk);
    int offset = skb_transport_offset(skb);
    int len = skb->len - offset;
```



```
wsum csum = 0;
/* 创建
```

UDP报文头部

```
*/
uh = udp_hdr(skb);
uh->source = inet->inet_sport;
uh->dest = fl4->fl4_dport;
uh->len = htons(len);
uh->check = 0;
/* 如果是轻量级
```

UDP协议，则调用相应的校验和计算函数。

想了解更多是

UDP Lite. 请自行

wiki.

```
*/
if (is_udplite)
    csum = udplite_csum(skb);
/* 禁止了
```

UDP校验和

```
*/
else if (sk->sk_no_check == UDP_CSUM_NOXMIT) {
    skb->ip_summed = CHECKSUM_NONE;
    goto send;
} else if (skb->ip_summed == CHECKSUM_PARTIAL) {
    /* 硬件支持校验和的计算
```

```
*/
    udp4_hwcsum(skb, fl4->saddr, fl4->daddr);
    goto send;
} else {
    /* 一般情况下的校验和计算
```

```
*/
    csum = udp_csum(skb);
}
/* 计算
```

UDP的校验和，需要考虑伪首部

```
*/
uh->check = csum_tcpudp_magic(fl4->saddr, fl4->daddr, len,
                               sk->sk_protocol, csum);
/* 如果校验和为
```

0，则需要将其设置为

0xFFFF. 因为

UDP的零校验和，有特殊的含义，表示没有校验和。

```
*/
if (uh->check == 0)
    uh->check = CSUM_MANGLED_0;
send:
/* 发送
```

IP数据包

```
*/
err = ip_send_skb(skb);
if (err) {
    if (err == -ENOBUFS && !inet->reovrr) {
        UDP_INC_STATS_USER(sock_net(sk),
                           UDP_MIB_SNDBUFFERRORS, is_udplite);
        err = 0;
    }
} else
    UDP_INC_STATS_USER(sock_net(sk),
                       UDP_MIB_OUTDATAGRAMS, is_udplite);
return err;
}
```

至此，UDP已经完成了自己的工作，后面的发送工作将交由IP层来负责。

在没有阅读内核源码时，我相信绝大多数的读者都会认为在使用UDP套接字时，每一次调用send都会产生一个UDP报文。事实上，在一般的项目中，UDP套接字确实也是这样使用的。然而通过阅读源码，我

13.4 TCP数据包的发送流程

13.3节追踪了UDP数据包的发送流程，本节要学习另外一个重要的传输层协议，TCP数据包的发送流程。

TCP的sendmsg操作函数为tcp_sendmsg，代码如下：

```
int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t size)
{
    struct iovec *iov;
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    int iovlen, flags;
    int mss_now, size_goal;
    int sg, err, copied;
    long timeo;
    lock_sock(sk);
    flags = msg->msg_flags;
    /* 根据标志，确定发送消息的超时时间；

    如果设置了

    MSG_DONTWAIT，则超时时间为

    0。

    若没有设置

    MSG_DONTWAIT，则使用套接字的超时时间。

    */
    timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
    /*
    套接字只有处于已连接（

    ESTABLISHED）和等待关闭（

    CLOSE_WAIT）的状态下，才能直接发送数据。

    已连接状态不用多说，等待关闭状态是指收到对端关闭

    (FIN)数据包，但本端应用还没有关闭连接时，这

    时仍然可以发送数据。

    在

    TCP协议中，发送

    FIN，表示本端不会再发送数据。

    */
    if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
        /* 等待连接建立，若失败则返回出错

    */
        if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
            goto out_err;
    /* 清除

    SOCK_ASYNC_NOSPACE标志

    */
    clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
    /* 得到当前的

    MSS长度和数据包的最大长度

    */
    mss_now = tcp_send_mss(sk, &size_goal, flags);
    /* 准备开始发送，获得用户的数据向量地址及长度
```

```

*/
    iovlen = msg->msg_iovlen;
    iov = msg->msg_iov;
    copied = 0;
    err = -EPIPE;
    /* 错误检查

*/

*/if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
    goto out_err;
/* 判断出口路由是否支持分数据聚合功能

*/

sg = sk->sk_route_caps & NETIF_F_SG;
/* 逐个发送数据段

*/

while (--iovlen >= 0) {
    /* 得到该数据段的长度及起始地址

*/

    size_t seglen = iov->iov_len;
    unsigned char __user *from = iov->iov_base;
    iov++;
    /* 循环以保证本数据段的数据全部被发送

*/

    while (seglen > 0) {
        int copy = 0;
        /* 获得数据包的最大长度

*/

        int max = size_goal;
        /* 获得发送队列尾部的

skb, 查看是否还有剩余空间

*/

        skb = tcp_write_queue_tail(sk);
        if (tcp_send_head(sk)) {
            if (skb->ip_summed == CHECKSUM_NONE)
                max = mss_now;
            /* 得到本次需要复制的长度

*/

            copy = max - skb->len;
        }
        /* 本

skb的数据长度已经超过了最大长度, 需要申请新的

skb */
        if (copy <= 0) {
            new_segment:
            /* 检查发送缓冲是否已经超出了限制

*/

            if (!sk_stream_memory_free(sk)) {
                /* 发送缓冲占用内存过多, 需要等待

*/

                goto wait_for_sndbuf;
            }
            /* 申请新的

skb */
            skb = sk_stream_alloc_skb(sk,
                select_size(sk, sg),
                sk->sk_allocation);
            if (!skb) {
                /* 若分配失败, 则需要等待

*/

                goto wait_for_memory;
            }
            /* 检查硬件是否支持校验和

*/

            if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
                skb->ip_summed = CHECKSUM_PARTIAL;
            /* 加入套接字的发送队列

*/

            skb_ensure_ownership(skb);
            copy = size_goal;
            max = size_goal;
        }
        /* 复制长度不能超过数据长度

*/

        if (copy > seglen)
            copy = seglen;
        /* 判断

```

skb的线性空间是否还有空闲

```
*/
    if (skb_availroom(skb) > 0) {
        /* 调整复制长度，不能超过空闲的空间长度

*/
    copy = min_t(int, copy, skb_availroom(skb));
    /* 将数据复制到
```

skb的空闲空间中

```
*/
    err = skb_add_data_nocache(sk, skb, from, copy);
    if (err)
        goto do_fault;
} else {
    /* 如果该
```

skb没有足够的空闲的线性空间，则把数据复制到分散聚合页中

```
*/
    int merge = 0;
    /* 获得数据的分片个数

*/
    int i = skb_shinfo(skb)->nr_frags;
    /* 获得套接字使用的页

*/
    struct page *page = TCP_PAGE(sk);
    /* 获得该页已使用的偏移

*/
    int off = TCP_OFF(sk);
    /* 判断数据包是否可以和最后一个分片聚合

*/
    if (skb_can_coalesce(skb, i, page, off) &&
        off != PAGE_SIZE) {
        /* 若可以聚合，则设置
```

merge标志

```
*/
    merge = 1;
} else if (i == MAX_SKB_FRAGS || !sg) {
    /* 已经达到分片上限，或者网络设备不支持分散聚合。这时不能再向分片增加任

    何数据了。

*/
    /*
    为了给新数据腾出空间，需要将老数据尽快发送出去。
```

因此设置

PUSH标志，并更新

pushed_seq. 然后跳转到

new_segment，并申请新的

skb.

```
*/
    tcp_mark_push(tp, skb);
    goto new_segment;
} else if (page) {
    /* 该页已满
```

```
*/
    if (off == PAGE_SIZE) {
        put_page(page);
        TCP_PAGE(sk) = page = NULL;
        off = 0;
    }
    else
        off = 0;
    /* 再次检查复制长度，不能超过该页的空闲长度
```

```

*/
    if (copy > PAGE_SIZE - off)
        copy = PAGE_SIZE - off;
    /* 增加发送缓存内存占用, 若超出限制, 则需要等待

*/

    if (!sk_wmem_schedule(sk, copy))
        goto wait_for_memory;
    /* 若没有可用的页, 则申请新的页

*/

    if (!page) {
        /* Allocate new cache page. */
        if (!(page = sk_stream_alloc_page(sk)))
            goto wait_for_memory;
    }
    /* 将数据复制到页的相应位置

*/

    err = skb_copy_to_page_nocache(sk, from, skb,
                                   page, off, copy);
    if (err) {
        /*
         * 即使复制失败, 如果该页是新申请的, 也应该让套接字拥有该页, 以供未来使用。

        */
        if (!TCP_PAGE(sk)) {
            TCP_PAGE(sk) = page;
            TCP_OFF(sk) = 0;
        }
        goto do_error;
    }
    /* Update the skb. */
    if (merge) {
        /*
         * 若本次数据可以和最后一个分片合并, 则更新最后一个分片的长度

        */
        skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
    } else {
        /* 这是新的分片, 需要为这个分片初始化一些页信息

    */
        skb_fill_page_desc(skb, i, page, off, copy);
        if (TCP_PAGE(sk)) {
            /* 该分页是之前分配的, 因此增加引用即可

        */
            get_page(page);
        } else if (off + copy < PAGE_SIZE) {
            /* 若该分页是新分配的, 但还未用完, 则增加引用, 并将其设置为套接字的

            发送页, 以便未来使用

        */
            TCP_PAGE(sk) = page;
        }
        /* 更新套接字的发送偏移量

    */
        TCP_OFF(sk) = off + copy;
    }
    /* 若无须复制任何数据, 则清除

PUSH标志

*/

    if (!copied)
        TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;
    /* 更新各种序列号

*/

    tp->write_seq += copy;
    TCP_SKB_CB(skb)->end_seq += copy;
    skb_shinfo(skb)->gso_segs = 0;
    /* 更新复制信息

*/

    from += copy;
    copied += copy;
    /* 判断是否完成了所有的数据拷贝

*/

    if ((seglen == copy) == 0 && iovlen == 0)
        goto out;
    /* 如果数据包的长度小于限制, 或者设置了

MSG_OOB标志, 则继续向该数据包增加数据

```

```

*/
    if (skb->len < max || (flags & MSG_OOB))
        continue;
    /* 如果当前序列号超过上次

```

push的序列号加上通告窗口的一半，则需要将本次数据包尽快发

送出去

```

*/
    if (forced_push(tp)) {
        /* 将本数据包设置上

```

PUSH标志，并更新

push序列号

```

*/
    tcp_mark_push(tp, skb);
    /* 将所有未送的数据包全都发送出去

```

```

*/
    tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
} else if (skb == tcp_send_head(sk)) {
    /* 如果套接字上只有当前这个数据包，就发送这一个数据包

```

```

*/
    tcp_push_one(sk, mss_now);
}
continue;
/* 等待发送缓存

```

```

*/
wait_for_sndbuf:
    /* 设置没有发送缓存的标志

```

```

*/
    set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
/* 等待内存

```

```

*/
wait_for_memory:
    /* 判断是否已经复制了部分数据

```

```

*/
    if (copied) {
        /* 去掉

```

MSG_MORE标志，表示尽快将复制的数据发送出去

```

*/
    tcp_push(sk, flags & ~MSG_MORE, mss_now, TCP_NAGLE_PUSH);
}
/* 等待空闲内存，可能进入睡眠状态

```

```

*/
    if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
        goto do_error;
    /* 有了空闲内存，但

```

MSS可能已经发生了变化，所以需要重新获取

```

MSS */
    mss_now = tcp_send_mss(sk, &size_goal, flags);
}
/* out是正常退出路径

```

```

*/
out:
    /* 如果成功复制了数据，则调用

```

tcp_push将数据包发送出去，但不保证立刻就发送

```

*/
    if (copied)
        tcp_push(sk, flags, mss_now, tp->nonagle);
    /* 释放套接字，返回发送的字节数

```

```

*/
    release_sock(sk);
    return Copied;
    /* 复制用户数据错误

```

```

*/
do_fault:
/* 如果当前

skb的数据长度为

0, 则需要从套接字的发送队列中将其删除, 并释放该

skb */
if (!skb->len) {
    tcp_unlink_write_queue(skb, sk);
    /* It is the one place in all of TCP, except connection
     * reset, where we can be unlinking the send_head.
     */
    tcp_check_send_head(sk, skb);
    sk_wmem_free_skb(sk, skb);
}
do_error:
/* 若出错时已经复制了部分数据, 则将已经复制的数据发送出去

*/
if (copied)
    goto out;
out_err:
/* 若没有复制任何数据, 则获取错误值, 释放套接字并返回错误

*/
err = sk_stream_error(sk, flags, err);
release_sock(sk);
return err;
}

```

因为TCP是一种流协议, 所以使用tcp_sendmsg发送数据时, 内核只是将数据包追加到套接字的发送队列中。真正发送数据的时刻, 则是由TCP协议来控制的, 套接字只能做出指示。tcp_sendmsg函数:

```

void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,
                              int nonagle)
{
    /* 如果套接字是关闭状态, 则直接返回

*/
    if (unlikely(sk->sk_state == TCP_CLOSE))
        return;
    /* tcp_write_xmit用于将

TCP报文发送到网络上

*/
    if (tcp_write_xmit(sk, cur_mss, nonagle, 0, GFP_ATOMIC)) {
        /* 如果没有要发送的数据, 则重置零窗口探测定时器

*/
        tcp_check_probe_timer(sk);
    }
}

```

下面进入tcp_write_xmit, 代码如下:

```

static int tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
                          int push_one, gfp_t gfp)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    unsigned int tso_segs, sent_pkts;
    int cwnd_quota;
    int result;
    sent_pkts = 0;
    /* 如果不是

push_one (即只发送一个数据包), 则进行

MTU探测

*/
    if (!push_one) {
        /* 进行

MTU探测

*/
        result = tcp_mtu_probe(sk);
        /* 若返回为

0, 则需要等待探测结果, 因此不能发送数据包。

*/
        if (!result) {
            return 0;
        } else if (result > 0) {

```

```
        sent_pkts = 1;
    }
}
/* 将发送队列中的数据包包，循环发送出去
```

```
*/
while ((skb = tcp_send_head(sk))) {
    unsigned int limit;
    /* 初始化这个数据包的
```

TSO状态。

TSO是

TCP Segment Offload的缩写。当

TCP发送数据时，需

要将数据拆分成

MSS大小的数据包（即多个

skb），然后再增加

TCP首部、

IP首部即可、计算校

验和等。而当网卡支持

TSO时，内核只需要增加

TCP首部即可，其余工作都交由网卡来处理。

```
*/
tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
BUG_ON(!tso_segs);
/* 检查拥塞窗口。若为
```

0，则不能发送

```
*/
cwnd_quota = tcp_cwnd_test(tp, skb);
if (!cwnd_quota)
    break;
/* 检查发送窗口。若为
```

0，则不能发送

```
*/
if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
    break;
if (tso_segs == 1) {
    /*
    只有一个
```

TSO数据段，进行

nagle算法检查。若返回

0，则不发送

```
*/
if (unlikely(!tcp_nagle_test(tp, skb, mss_now,
                             (tcp_skb_is_last(sk, skb) ?
                              nonagle : TCP_NAGLE_PUSH))))
    break;
} else {
    /* 多个
```

TSO数据段

```
*/
/* 如果没有设置
```


push_one标志并且

TSO发送算法判断推迟发送，则暂不发送这个数据包

```
*/
    if (!push_one && tcp_tso_should_defer(sk, skb))
        break;
    }
    limit = mss_now;
    /* 当
```

TSO分段多于一个并且不是紧急模式

,则利用

MSS和可分段的个数（拥塞窗口和

GSO最大分段数

量之间的最小值）得到数据的最长限制

```
*/
    if (tso_segs > 1 && !tcp_urg_mode(tp))
        limit = tcp_mss_split_point(sk, skb, mss_now,
                                     min_t(unsigned int,
                                             cwnd_quota,
                                             sk->sk_gso_max_segs));
    /*
    若数据长度大于限制，则需要分片。
```

若分片失败，则暂不发送这个数据包。

```
*/
    if (skb->len > limit &&
        unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
        break;
    /* 更新
```

TCP控制块的时间戳

```
*/
    TCP_SKB_CB(skb)->when = tcp_time_stamp;
    /* 发送数据包
```

```
*/
    if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
        break;
    /* 处理发送新数据事件，如调整发送队列，则重置重传定时器等
```

```
*/
    tcp_event_new_data_sent(sk, skb);
    /* 更新小包（即小于
```

MSS大小）的发送时间

```
*/
    tcp_minshall_update(tp, mss_now, skb);
    /* 更新发送数据包的数量
```

```
*/
    sent_pkts += tcp_skb_pcount(skb);
    /* 如果设置了
```

push_one标志，则只发送一个数据包，因此可直接退出

```
*/
    if (push_one)
        break;
    }
    /* 如果当前处于拥塞恢复的状态下，则增加这个状态下的发包数量
```

```
*/
    if (inet_csk(sk)->icsk_ca_state == TCP_CA_Recovery)
        tp->prr_out += sent_pkts;
    /* 如果发送了数据，则校验发送拥塞窗口
```

```
*/
    if (likely(sent_pkts)) {
        tcp_cwnd_validate(sk);
```

```
        return 0;
    }
    return !tp->packets_out && tcp_send_head(sk);
}
```

继续往下跟踪TCP的发送函数tcp_transmit_skb，代码如下：

```
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
                           gfp_t gfp_mask)
{
    const struct inet_connection_sock *icsk = inet_csk(sk);
    struct inet_sock *inet;
    struct tcp_sock *tp;
    struct tcp_skb_cb *tcb;
    struct tcp_out_options opts;
    unsigned tcp_options_size, tcp_header_size;
    struct tcp_md5sig_key *md5;
    struct tcphdr *th;
    int err;
    BUG_ON(!skb || !tcp_skb_pcount(skb));
    /* 判断拥塞控制算法是否需要运行时间采样。如果需要，则获取当前时间

*/
    if (icsk->icsk_ca_ops->flags & TCP_CONG_RTT_STAMP)
        net_timestamp(skb);
    /* 判断是否需要克隆这个数据包
```

```
*/
    if (likely(clone_it)) {
        /*
        如果该数据包已经被克隆了，则需要复制
```

SKB的私有部分。

如未克隆，则直接克隆该数据包

```
*/
    if (unlikely(skb_cloned(skb)))
        skb = pskb_copy(skb, gfp_mask);
    else
        skb = skb_clone(skb, gfp_mask);
    if (unlikely(!skb))
        return -ENOMEM;
}
inet = inet_sk(sk);
tp = tcp_sk(sk);
tcb = TCP_SKB_CB(skb);
memset(&opts, 0, sizeof(opts));
/* 根据
```

TCP包的类型计算

TCP选项部分的大小

```
*/
    if (unlikely(tcb->tcp_flags & TCPHDR_SYN))
        tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);
    else
        tcp_options_size = tcp_established_options(sk, skb, &opts,
                                                    &md5);
    /* 得到完整的
```

TCP首部大小

```
*/
    tcp_header_size = tcp_options_size + sizeof(struct tcphdr);
    /* 判断是否有未确认的数据包
```

```
*/
    if (tcp_packets_in_flight(tp) == 0) {
        /* 通知开始发送事件
```

```
*/
        tcp_ca_event(sk, CA_EVENT_TX_START);
        /* 若设置了
```

ooo_okay标志，则表明可以改变发送队列。参见内核的

XPS发送机制

```
*/
        skb->ooo_okay = 1;
    } else {
        /* 若清除
```

ooo_okay标志，则表示不能改变发送队列。参见内核的

XPS发送机制

```
*/
    skb->ooo_okay = 0;
}
/* 在
```

skb中为

TCP首部申请空间

```
*/
    skb_push(skb, tcp_header_size);
/* 设置
```

TCP首部的起始位置

```
*/
    skb_reset_transport_header(skb);
/* 将数据包包入到发送队列中
```

```
*/
    skb_set_owner_w(skb, sk);
/* 构建
```

TCP首部, 并计算校验和

```
*/
    th = tcp_hdr(skb);
    th->source      = inet->inet_sport;
    th->dest         = inet->inet_dport;
    th->seq          = htonl(tcb->seq);
    th->ack_seq      = htonl(tp->rcv_nxt);
    *((__be16 *)th) + 6 = htons((tcp_header_size >> 2) << 12) |
        tcb->tcp_flags;
    if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
        /* RFC1323: The window in SYN & SYN/ACK segments
         * is never scaled.
         */
        th->window = htons(min(tp->rcv_wnd, 65535));
    } else {
        th->window = htons(tcp_select_window(sk));
    }
    th->check      = 0;
    th->urg_ptr     = 0;
    /* The urg mode check is necessary during a below snd_una win probe */
    if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
        if (before(tp->snd_up, tcb->seq + 0x10000)) {
            th->urg_ptr = htons(tp->snd_up - tcb->seq);
            th->urg = 1;
        } else if (after(tcb->seq + 0xFFFF, tp->snd_nxt)) {
            th->urg_ptr = htons(0xFFFF);
            th->urg = 1;
        }
    }
    /* 构建
```

TCP选项

```
*/
    tcp_options_write((__be32 *) (th + 1), tp, &opts);
/* 如果不是
```

SYN数据包包, 则尝试设置

ECN状态

```
*/
    if (likely((tcb->tcp_flags & TCPHDR_SYN) == 0))
        TCP_ECN_send(sk, skb, tcp_header_size);
#ifdef CONFIG_TCP_MD5SIG
/* 计算
```

TCP MD5 签名

```
*/
    if (md5) {
        sk_nocaps_add(sk, NETIF_F_GSO_MASK);
        tp->af_specific->calc_md5_hash(opts.hash_location,
                                       md5, sk, NULL, skb);
    }
#endif
/* 计算
```

TCP的校验和

```
*/
    icsk->icsk_af_ops->send_check(sk, skb);
/* 如果有
```

ACK标志, 则发送

ACK事件通知

```
*/
if (likely(tcb->tcp_flags & TCPHDR_ACK))
    tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
/* 如果数据包长度大于
```

TCP首部，那么自然是有

TCP数据的，所以数据将发送事件通知

```
*/
if (skb->len != tcp_header_size)
    tcp_event_data_sent(tp, skb);
/* 增加
```

TCP发送数据包的统计计数

```
*/
if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
    TCP_ADD_STATS(sock_net(sk), TCP_MIR_OUTSEGS,
                  tcp_skb_pcount(skb));
/* 调用
```

ip_queue_xmit发送数据报文

```
*/
err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);
if (likely(err <= 0))
    return err;
/* 判断是否需要进入拥塞窗口来恢复状态
```

```
*/
tcp_enter_cwr(sk, 1);
/* 因为
```

NET_XMIT_CN 返回值，不能被看作发送错误。

所以对于发送返回的错误，需要调用

net_xmit_eval来屏蔽该错误

```
*/
return net_xmit_eval(err);
}
```

至此，TCP也完成了自己的工作，IP层将负责后面的数据包发送工作。

13.5 IP数据包的发送流程

前面两节分别分析学习了UDP和TCP的发送流程。它们在完成各自的工作，并构建对应的首部以后，就将数据包传递给了IP网络层。一般情况下，UDP和TCP使用不同的网络层接口函数来将数据包传递给网络层。下文将分别对UDP和TCP进行详细介绍。

13.5.1 ip_send_skb源码分析

UDP调用ip_send_skb将数据包传给网络层，下面是其源码分析：

```
int ip_send_skb(struct sk_buff *skb)
{
    struct net *net = sock_net(skb->sk);
    int err;
    /* ip_local_out为本机发送

    IP数据包函数

    */
    err = ip_local_out(skb);
    if (err) {
        /* 发送错误

    */
        if (err > 0) {
            /* 利用

net_xmit_errno转换发送错误值

    */
            err = net_xmit_errno(err);
        }
        if (err)
            IP_INC_STATS(net, IPSTATS_MIB_OUTDISCARDS);
        return err;
    }
}
```

进入ip_local_out，代码如下：

```
int ip_local_out(struct sk_buff *skb)
{
    int err;
    /* Linux内核代码充斥了大量的封装函数，如

    ip_local_out、

    __ip_local_out、等等

    */
    /* 检查

netfilter在本机的发送路径

    */
    err = __ip_local_out(skb);
    /* 若
```

err为

1, 则表示通过了

netfilter检查

```
*/
    if (likely(err == 1)) {
        /* 调用路由输出函数, 发送数据包

*/
        err = dst_output(skb);
    }
    return err;
}
```

进入__ip_local_out, 代码如下:

```
int __ip_local_out(struct sk_buff *skb)
{
    /* 得到
```

IP首部

```
*/
    struct iphdr *iph = ip_hdr(skb);
    /* 计算
```

IP报文的总长度

```
*/
    iph->tot_len = htons(skb->len);
    /* 计算
```

IP报文的校验和

```
*/
    ip_send_check(iph);
    /* 检查
```

netfilter的

localout路径

```
*/
```

```
return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT, skb, NULL,  
              skb_dst(skb)->dev, dst_output);  
}
```

Netfilter的源码并不复杂，并且由于与当前主题的相关度并不高，所以在此就不对Netfilter的相关代码进行跟踪分析了。我们可以假设在没有使用Netfilter或没有对应的规则时，nf_hook会返回1。这样发送数据包的关键就在于dst_output函数了。

dst_output函数的实现为skb_dst(skb)->output(skb)。其中skb_dst(skb)为这个数据包找到的路由缓存，output为其实现发送功能的函数指针。这里面又涉及一个内核常用的编程技巧，利用函数指针将两个层次或功能模块进行隔离解耦。对于内核来说，无论是要发送出去的数据包，还是接收到的数据，在构建完IP报文后，都要通过查找路由来确定下一步的流程。而通过查找到的路由缓存的input和output函数指针，就可以确定后续的处理。内核提供了几个公共的路由输出函数，应用于不同的场景的路由，如dst_discard用于失效的路由，ip_rt_bug用于非预期的输出，ip_mc_output用于本机多播输出，而ip_output则用于本机向外发送数据包。

因此，对于本机发出的数据包，其路由输出函数即为ip_output，它的实现非常简单，代码如下：

```
int ip_output(struct sk_buff *skb)  
{  
    /* 得到发送设备  
  
    */  
    struct net_device *dev = skb_dst(skb)->dev;  
    /* 增加  
  
    IP数据包发送统计计数  
  
    */  
    IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUT, skb->len);  
    /* 设置数据包的出口设备  
  
    */  
    skb->dev = dev;  
    /* 设置数据包的协议为  
  
    IP协议  
  
    */  
    skb->protocol = htons(ETH_P_IP);  
    /* 进行  
  
    Netfilter在  
  
    POST_ROUTING上的检查
```



```
*/
return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, skb, NULL, dev,
                    ip_finish_output,
                    !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

通过了Netfilter在POST ROUTING上的检查后，数据包将进入ip_finish_output，代码如下：

```
static int ip_finish_output(struct sk_buff *skb)
{
#ifdef CONFIG_NETFILTER && defined(CONFIG_XFRM)
    /* 如果是路由缓存表示需要变换

*/
    if (skb_dst(skb)->xfrm != NULL) {
        /* 设置上重新选路的标志

*/
        IPCB(skb)->flags |= IPSKB_REROUTED;
        return dst_output(skb);
    }
#endif
    /* 如果数据包长度超过
```

MTU，并且数据包不是

GSO数据包

```
*/
if (skb->len > ip_skb_dst_mtu(skb) && !skb_is_gso(skb)){
    /* 执行
```

IP分片，因不是本文重点，故略过

```
*/
    return ip_fragment(skb, ip_finish_output2);
}
else {
    /* 进入真正的三层发送函数

*/
    return ip_finish_output2(skb);
}
```

继续进入ip_finish_output2，代码如下：

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
    struct dst_entry *dst = skb_dst(skb);
    struct rtable *rt = (struct rtable *)dst;
    struct net_device *dev = dst->dev;
    unsigned int hh_len = LL_RESERVED_SPACE(dev);
    struct neighbour *neigh;
    /* 根据路由类型是多播或广播，来增加相应的计数
```

```

*/
if (rt->rt_type == RTN_MULTICAST) {
    IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUTMCAST, skb->len);
} else if (rt->rt_type == RTN_BROADCAST)
    IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUTBCAST, skb->len);
/* 检查数据包的首部是否还有存放二层首部的空间

```

```

*/
if (unlikely(skb_headroom(skb) < hh_len && dev->header_ops)) {
    struct sk_buff *skb2;
    /* 重新申请一个足够空间的

```

```

skb */
    skb2 = skb_realloc_headroom(skb, LL_RESERVED_SPACE(dev));
    if (skb2 == NULL) {
        kfree_skb(skb);
        return -ENOMEM;
    }
    /* 如果原数据包属于某个套接字，则将新数据包也设置成归属于这个套接字

```

```

*/
    if (skb->sk)
        skb_set_owner_w(skb2, skb->sk);
    /* 释放原数据包的内存空间，让原数据包的

```

skb指针指向新数据包的内存空间

```

*/
    kfree_skb(skb);
    skb = skb2;
}
rcu_read_lock();
/* 获得路由的

```

neighbour信息

```

*/
neigh = dst_get_neighbour(dst);
if (neigh) {
    /* 调用

```

neighbour层的输出接口。是否能够立刻发送，依赖于

neighbour的状态

```

*/
    int res = neigh_output(neigh, skb);
    rcu_read_unlock();
    return res;
}
rcu_read_unlock();
/* 若该路由没有

```

neighbour的信息，则输出报错

```
*/
if (net_ratelimit())
    printk(KERN_DEBUG "ip_finish_output2: No header cache and no neighbour!\n");
kfree_skb(skb);
return -EINVAL;
}
```

13.5.2 ip_queue_xmit源码分析

13.5.1节的ip_send_skb是UDP调用的IP层的输出接口，而TCP调用的IP层输出接口则为ip_queue_xmit。下面来看看相应的源码：

```
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
    struct sock *sk = skb->sk;
    struct inet_sock *inet = inet_sk(sk);
    struct ip_options_rcu *inet_opt;
    struct flowi4 *fl4;
    struct rtable *rt;
    struct iphdr *iph;
    int res;
    /* 判断数据包是否有路由，如果已经有了，就直接跳到
```

```
packet_routed */
    rcu_read_lock();
    inet_opt = rcu_dereference(inet->inet_opt);
    fl4 = &fl->u.ip4;
    rt = skb_rtable(skb);
    if (rt != NULL)
        goto packet_routed;
    /* 从套接字获得合法的路由（需要检查是否过期）
```

```
*/
    rt = (struct rtable *) __sk_dst_check(sk, 0);
    if (rt == NULL) {
        __be32 daddr;
        daddr = inet->inet_daddr;
        /* 如果有
```

IP 严格路由选项，则使用选项中的地址作为目的地址进行路由查询

```
*/
    if (inet_opt && inet_opt->opt.srr)
        daddr = inet_opt->opt.faddr;
    /* 进行路由查找
```

```
*/
    rt = ip_route_output_ports(sock_net(sk), fl4, sk,
                               daddr, inet->inet_saddr,
                               inet->inet_dport,
                               inet->inet_sport,
                               sk->sk_protocol,
                               RT_CONN_FLAGS(sk),
                               sk->sk_bound_dev_if);
    if (IS_ERR(rt))
        goto no_route;
    /* 根据路由的接口的特性设置套接字特性
```

```
*/
    sk_setup_caps(sk, &rt->dst);
}
/* 给数据包设置路由
```

```
*/
    skb_dst_set_noref(skb, &rt->dst);
packet_routed:
    /* 如果有
```

IP严格路由选项

```
*/
if (inet_opt && inet_opt->opt.is_strictroute && fl4->daddr != rt->rt_gateway)
    goto no_route;
/* 分配
```

IP首部和选项空间

```
*/
skb_push(skb, sizeof(struct iphdr) + (inet_opt ? inet_opt->opt.optlen : 0));
/* 设置
```

IP首部位置

```
*/
skb_reset_network_header(skb);
/* 得到数据包
```

IP首部的指针

```
*/
iph = ip_hdr(skb);
/* 构建
```

IP首部

```
*/
*((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
/* 如不能分片, 则在
```

IP首部设置

IP_DF标志

```
*/
if (ip_dont_fragment(sk, &rt->dst) && !skb->local_df)
    iph->frag_off = htons(IP_DF);
else
    iph->frag_off = 0;
iph->ttl = ip_select_ttl(inet, &rt->dst);
iph->protocol = sk->sk_protocol;
iph->saddr = fl4->saddr;
iph->daddr = fl4->daddr;
/* Transport layer set skb->h.foo itself. */
/* 构建
```

IP选项

```
*/
```

```

if (inet_opt && inet_opt->opt.optlen) {
    iph->ihl += inet_opt->opt.optlen >> 2;
    ip_options_build(skb, &inet_opt->opt, inet->inet_daddr, rt, 0);
}
/* 选择合适的

IP identifier */
ip_select_ident_more(iph, &rt->dst, sk,
    (skb_shinfo(skb)->gso_segs ?: 1) - 1);
/* 根据套接字选项, 设置数据包的优先级和标记

*/
skb->priority = sk->sk_priority;
skb->mark = sk->sk_mark;
/* 发送数据包

*/
res = ip_local_out(skb);
rcu_read_unlock();
return res;
no_route:
rcu_read_unlock();
IP_INC_STATS(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
kfree_skb(skb);
return -EHOSTUNREACH;
}

```

ip_queue_xmit最终也是调用ip_local_out发送本机的数据包。该函数已经在前面跟踪分析过了，所以在此就不再重复了。

13.6 底层模块数据包的发送流程

13.5节分析了IP网络层的数据包的发送流程，并最终跟踪到其调用邻居模块的发送接口。为什么内核会有一个邻居模块呢？本质上数据包的发送和接收都依赖于数据链路层（二层）的地址即硬件地址，网卡只接受二层目的地址为自己地址的数据包（或者多播、广播地址）。所谓的IP地址（三层）只是一个逻辑地址，其实际用途是用来寻径的。那么内核在发送数据包的时候，就需要填充正确的二层硬件地址才能将数据包成功地发送出去。这里就有了一个需求，即需要将三层网络地址“映射”为正确的二层硬件地址。对于IPv4来说，这是由ARP协议来实现的，而对于IPv6来说，其邻居发现协议是由ICMPv6来实现的。因此，对于内核来说，一方面是为了屏蔽不同的邻居协议的实现细节；另一方面，使用同一个邻居模块，对外可以保证相同的邻居状态机和一致的接口。

13.5节中，二层数据包的发送接口为`neigh_output`，其源码如下：

```
static inline int neigh_output(struct neighbour *n, struct sk_buff *skb)
{
    struct hh_cache *hh = &n->hh;
    /*
     * 若邻居状态为连接状态：永久邻居，不需要
     *
     * ARP，可到达三种情况，
     *
     * 并且存在硬件地址，则直接调用
     *
     * neigh_hh_output来发送。
     *
     * 不然则通过邻居的输出函数发送—会根据邻居状态使用不同的接口。
     *
     */
    if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)
        return neigh_hh_output(hh, skb);
    else
        return n->output(n, skb);
}
```

先跟踪第一种情况，来查看`neigh_hh_output`的代码：

```
static inline int neigh_hh_output(struct hh_cache *hh, struct sk_buff *skb)
{
    unsigned seq;
    int hh_len;
    /*
     * 使用
     *
     * seqlock读取硬件地址。
     */
}
```

seqlock一般用在频繁读操作，偶尔写操作的情况下。读操作并不会真正地上锁，因此不会阻塞其他读操

作和写操作，并通过序号来保证读出数据的完整性；写操作会使用

spinlock来保证同一时间只有一个写

操作。

```
*/
do {
    int hh_alen;
    seq = read_seqbegin(&hh->hh_lock);
    hh_len = hh->hh_len;
    hh_alen = HH_DATA_ALIGN(hh_len);
    memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
} while (read_seqretry(&hh->hh_lock, seq));
/* 保存硬件地址

*/
skb_push(skb, hh_len);
/* 调用底层发送数据包接口

*/
return dev_queue_xmit(skb);
}
```

下面再来分析邻居在不同状态下的发送接口，在此，以IPv4的ARP协议为例来进行分析。

首先我们来看看NUD_NOARP状态，代码如下：

```
neigh->nud_state = NUD_NOARP;
neigh->ops = &arp_direct_ops;
neigh->output = neigh_direct_output;
```

在NUD_NOARP状态下，neigh的发送接口为neigh_direct_output，代码如下。

```
int neigh_direct_output(struct neighbour *neigh, struct sk_buff *skb)
{
    return dev_queue_xmit(skb);
}
```

这个函数“码如其名”，就是直接调用底层的发送接口。

再看看NUD_VALID状态和其余状态，代码如下：

```
if (dev->header_ops->cache)
    neigh->ops = &arp_hh_ops;
else
    neigh->ops = &arp_generic_ops;
```



```
if (neigh->nud_state & NUD_VALID)
    neigh->output = neigh->ops->connected_output;
else
    neigh->output = neigh->ops->output;
```

若网卡提供了首部缓存的功能，则邻居的操作函数为arp_hh_ops，不然则为arp_generic_ops。对于arp_generic_ops来说，若邻居状态为NUD_VALID，则输出函数为neigh_connected_output（与NUD_NOARP状态相同），其余状态则为neigh_resolve_output，代码如下：

```
int neigh_resolve_output(struct neighbour *neigh, struct sk_buff *skb)
{
    struct dst_entry *dst = skb_dst(skb);
    int rc = 0;
    if (!dst)
        goto discard;
    /* 根据具体的邻居发现协议，发送探测邻居数据包。对于
```

IPv4来说，就是

ARP请求。如果成功得到邻

居的地址，则返回成功（数值

0），不然则返回错误值

```
*/
if (!neigh_event_send(neigh, skb)) {
    /* 有了邻居即对端硬件地址，就可以发送数据包了

*/
    int err;
    struct net_device *dev = neigh->dev;
    unsigned int seq;
    /* 如果网卡有地址缓存功能，并且邻居模块没有对应的硬件地址，则调用网卡功能，填充二层硬件地址

*/
    if (dev->header_ops->cache && !neigh->hh.hh_len)
        neigh_hh_init(neigh, dst);
    /* 下面的代码与
```

neigh_hh_output类似，利用

seqlock在无锁的条件下，保证二层地址读取的完整性。

```
*/
do {
    __skb_pull(skb, skb_network_offset(skb));
    seq = read_seqbegin(&neigh->ha_lock);
    err = dev_hard_header(skb, dev, ntohs(skb->protocol),
                          neigh->ha, NULL, skb->len);
} while (read_seqretry(&neigh->ha_lock, seq));
/* 若成功读取了硬件地址，则调用底层发送函数，将数据包发送出去。
```

```

*/
    if (err >= 0)
        rc = dev_queue_xmit(skb);
    else
        goto out_kfree_skb;
}
out:
    return rc;
discard:
    NEIGH_PRINTK1("neigh_resolve_output: dst=%p neigh=%p\n",
        dst, neigh);
out_kfree_skb:
    rc = -EINVAL;
    kfree_skb(skb);
    goto out;
}

```

邻居模块除了相应的发送过程外，更为重要的是邻居模块状态变迁的状态机，以及邻居发现协议的实现。但是这两部分与本章的主题关联并不大，代码也不复杂，熟悉相关协议的读者可以很容易看懂该部分代码。

邻居模块调用的底层发送接口为`dev_queue_xmit`，实质上数据包会首先进入内核的TC模块的队列（默认情况下，网卡使用的TC队列为`PFIFO_FAST`队列），然后根据队列算法，让合适的数据包出队（之所以说合适的，是因为不同的TC队列，其出队的算法不同），并调用网卡的发送函数，将数据包发送到网卡。如果这时网卡满足发送条件，则数据包将会被真正地发送出去。若不满足发送条件，则将利用发送软中断，过段时间再进行下一轮尝试。

第14章 网络通信：数据报文的接收

第13章完成了对数据包发送流程的分析和学习，本章则要学习数据包的接收过程，同样也从应用层开始入手，然后深入到内核的实现代码，从而真正理解接收数据的接口。本章也是网络通信的最后一章。

14.1 系统调用接口

与发送类似，内核也提供了多个接收数据的系统调用接口，接口定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

与send类似，recv一般也是面向连接的套接字。原因在于，对于非面向连接的套接字来说，若使用recv接收数据，通过该接口将不能获得发送端的地址，也就是说不知道这个数据是谁发过来的。所以，如果使用者不关心发送端信息，或者该信息可以从数据中获得，那么recv接口同样也可以用于非面向连接的套接字。再来看看recvfrom，它会通过额外的参数src_addr和addrlen，来获得发送方的地址，其中需要注意的是addrlen，它既是输入值又是输出值。最后是recvmsg，它与sendmsg一样，把接收到的数据和地址都保存在了msg中。其中msg.msg_name和msg.msg_len用于保存接收端地址，而msg.msg_iov用于保存接收到的数据。这三个系统调用与对应的发送接口一样，都支持设置标志位flags——都是比较现代的接口设计方法。

14.2 数据包从内核空间到用户空间的流程

第13章中，几个不同的发送数据包的系统调用，最终都是通过公共的函数`sock_sendmsg`来完成的。那么对于接收数据包的系统调用，我们相信它们也是殊途同归，最后会进入到一个公共的函数中。接下来，跟踪14.1节介绍的三个系统调用的实现，来证明我们的猜想。

首先是`recv`的源码：

```
asmlinkage long sys_recv(int fd, void __user *ubuf, size_t size,
                        unsigned flags)
{
    return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);
}
```

代码很简单，`recv`完全是通过调用`sys_recvfrom`来实现的，仅仅是将`sys_recvfrom`的最后两个参数设置为0而已。

那么接下来就进入`recvfrom`的源码：

```
SYSCALL_DEFINE6(recvfrom, int, fd, void __user *, ubuf, size_t, size,
                unsigned, flags, struct sockaddr __user *, addr,
                int __user *, addr_len)
{
    struct socket *sock;
    struct iovec iov;
    struct msghdr msg;
    struct sockaddr_storage address;
    int err, err2;
    int fput_needed;
    /* 限制读取字节长度的最大值为整数的最大值
```

```
    INT_MAX */
    if (size > INT_MAX)
        size = INT_MAX;
    /* 从文件描述符得到套接字结构
```

```
    */
    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (!sock)
        goto out;
    /* 控制信息清零
```

```
    */
    msg.msg_control = NULL;
    msg.msg_controllen = 0;
    /* 设置消息的数据段信息
```

```
    */
    msg.msg_iovlen = 1;
    msg.msg_iov = &iov;
    iov.iov_len = size;
    iov.iov_base = ubuf;
    /* 设置消息的存储地址信息
```

```
    */
    msg.msg_name = (struct sockaddr *)&address;
    msg.msg_namelen = sizeof(address);
    /* 如果套接字设置了
```

`O_NONBLOCK`标志，即非阻塞标志，则设置

`MSG_DONTWAIT`标志，表示此次接收消息，无须等待

```

*/
if (sock->file->f_flags & O_NONBLOCK)
    flags |= MSG_DONTWAIT;
/* 调用

sock_recvmsg接收数据

*/
err = sock_recvmsg(sock, &msg, size, flags);
/* 将地址信息复制到用户空间

*/
if (err >= 0 && addr != NULL) {
    err2 = move_addr_to_user((struct sockaddr *)&address,
                             msg.msg_namelen, addr, addr_len);
    if (err2 < 0)
        err = err2;
}
fput_light(sock->file, fput_needed);
out:
return err;
}

```

后面的调用流程则为sock_recvmsg→__sock_recvmsg→__sock_recvmsg_nosec。

下面跟踪第三个接收数据包的系统调用recvmsg，代码如下：

```

SYSCALL_DEFINE3(recvmsg, int, fd, struct msghdr __user *, msg,
                unsigned int, flags)
{
    int fput_needed, err;
    struct msghdr msg_sys;
    /* 从文件描述符

fd获得套接字

*/
    struct socket *sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (!sock)
        goto out;
    /* __sys_recvmsg用于实现接收数据

*/
    err = __sys_recvmsg(sock, msg, &msg_sys, flags, 0);
    /* 释放

fd引用（如果需要的话），这也是

fput_light与

fput的区别

*/
    fput_light(sock->file, fput_needed);
out:
    return err;
}

```

下面进入__sys_recvmsg，代码如下：

```

static int __sys_recvmsg(struct socket *sock, struct msghdr __user *msg,
                        struct msghdr *msg_sys, unsigned flags, int nsec)
{
    struct compat_msghdr __user *msg_compat =
        (struct compat_msghdr __user *)msg;
    struct iovec iovstack[UIO_FASTIOV];
    struct iovec *iov = iovstack;
    unsigned long cmsg_ptr;
    int err, iov_size, total_len, len;
    /* kernel mode address */
    struct sockaddr_storage addr;
    /* user mode address pointers */
    struct sockaddr __user *uaddr;
    int __user *uaddr_len;
    /* 将消息头从用户空间复制到内核空间 */

    /*
    if (MSG_CMSG_COMPAT & flags) {
        if (get_compat_msghdr(msg_sys, msg_compat))
            return -EFAULT;
    } else if (copy_from_user(msg_sys, msg, sizeof(struct msghdr)))
        return -EFAULT;
    err = -EMSGSIZE;
    /* 检查数据段的个数 */

    /*
    if (msg_sys->msg_iovlen > UIO_MAXIOV)
        goto out;
    /*
    为了避免频繁申请内存，内核在栈上申请了

UIO_FASTIOV大小的

iovec数组以供

iov使用。当数据段个数

超过

UIO_FASTIOV时，就需要动态申请内存。

    /*
    err = -ENOMEM;
    iov_size = msg_sys->msg_iovlen * sizeof(struct iovec);
    if (msg_sys->msg_iovlen > UIO_FASTIOV) {
        iov = sock_kmalloc(sock->sk, iov_size, GFP_KERNEL);
        if (!iov)
            goto out;
    }
    /* 验证用户传递的数据段参数和地址参数

    /*
    uaddr = (__force void __user *)msg_sys->msg_name;
    uaddr_len = COMPAT_NAMELEN(msg);
    if (MSG_CMSG_COMPAT & flags) {
        err = verify_compat_iovec(msg_sys, iov,
                                   (struct sockaddr *)&addr,
                                   VERIFY_WRITE);
    } else
        err = verify_iovec(msg_sys, iov,
                            (struct sockaddr *)&addr,
                            VERIFY_WRITE);
    if (err < 0)
        goto out_freeiov;
    total_len = err;
    cmsg_ptr = (unsigned long)msg_sys->msg_control;
    /* 确保消息标志中只有内核支持的两个标志

    /*
    msg_sys->msg_flags = flags & (MSG_CMSG_CLOEXEC|MSG_CMSG_COMPAT);
    /* 如果套接字为非阻塞，则设置标志位为不等待（非阻塞）

```

```

*/
if (sock->file->f_flags & O_NONBLOCK)
    flags |= MSG_DONTWAIT;
/* 根据安全检查标志，调用不同的接收函数，但最终都会调用到

sock_recvmmsg */
err = (nosec ? sock_recvmmsg_nosec : sock_recvmmsg)(sock, msg_sys,
                                                    total_len, flags);
if (err < 0)
    goto out_freeiov;
len = err;
/* 将发送端的地址复制到用户空间

*/
if (uaddr != NULL) {
    err = move_addr_to_user((struct sockaddr *)&addr,
                           msg_sys->msg_namelen, uaddr,
                           uaddr_len);
    if (err < 0)
        goto out_freeiov;
}
.....

.....

}

```

由上面的代码可以看出，内核提供的三个接收数据包的系统调用，最终确实如我们所期望的，都会走到一个共同的函数__sock_recvmmsg_nosec里。下面来看一下这个函数，代码如下：

```

static inline int __sock_recvmmsg_nosec(struct kiocb *iocb, struct socket *sock, struct msghdr *msg, size_t size, int flags)
{
    struct sock_iocb *si = kiocb_to_siocb(iocb);
    sock_update_classid(sock->sk);
    /* 设置套接字异步

IO信息

*/
    si->sock = sock;
    si->scm = NULL;
    si->msg = msg;
    si->size = size;
    si->flags = flags;
    /* 根据不同的套接字类型，调用不同的数据接收函数

*/
    return sock->ops->recvmmsg(iocb, sock, msg, size, flags);
}

```

根据上面的代码，后面的接收流程就要依赖于具体的协议实现了。

14.3 UDP数据包的接收流程

首先，我们来分析一下相对简单的UDP协议的数据包接收流程，代码如下：

```
int udp_recvmmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                 size_t len, int noblock, int flags, int *addr_len)
{
    struct inet_sock *inet = inet_sk(sk);
    /* 让
```

sin指向

msg_name, 用于保存发送端地址

```
*/
    struct sockaddr_in *sin = (struct sockaddr_in *)msg->msg_name;
    struct sk_buff *skb;
    unsigned int ulen, copied;
    int peeked;
    int err;
    int is_udplite = IS_UDPLITE(sk);
    bool slow;
    /* 若
```

addr_len不为

NULL, 即用户传递了地址长度参数。进入了具体的协议层, 已经可以明确地址的长度信息了。

```
*/
    if (addr_len)
        *addr_len = sizeof(*sin);
    /* 用户设置了
```

MSG_ERRQUEUE标志, 用于接收错误消息。因为这个应用并不广泛, 因此在此忽略这种情况, 不进入该函数。

```
*/
    if (flags & MSG_ERRQUEUE)
        return ip_rcv_error(sk, msg, len);
try_again:
    /* 接收了一个数据报文
```

```
*/
    skb = __skb_recv_datagram(sk, flags | (noblock ? MSG_DONTWAIT : 0),
                              &peeked, &err);
    /* 若没有收到报文, 则直接退出
```

```
*/
    if (!skb)
        goto out;
    /* 得到
```

UDP的数据长度

```

*/
ulen = skb->len - sizeof(struct udphdr);
/* 要复制的长度被初始化为用户指定的长度

```

```

*/
copied = len;
/* 若复制长度大于

```

UDP的数据长度，则调整复制长度为数据长度。若复制长度小于数据长度，则设置标志

MSG_TRUNC，表示数据发生了截断。

```

*/
if (copied > ulen)
    copied = ulen;
else if (copied < ulen)
    msg->msg_flags |= MSG_TRUNC;
/*
如果发生了数据截断，或者我们只需要部分覆盖的校验和，那么就在复制前进行校验。

```

```

*/
if (copied < ulen || UDP_SKB_CB(skb)->partial_cov) {
    /* 进行

```

UDP校验和校验

```

*/
    if (udp_lib_checksum_complete(skb))
        goto csum_copy_err;
}
/* 判断是否需要进行校验和校验

```

```

*/
if (skb_csum_unnecessary(skb)) {
    /* 若不需要进行校验，则直接复制数据包内容到

```

msg_iov中

```

*/
    err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr),
                                   msg->msg_iov, copied);
}
else {
    /* 复制数据包内容的同时，进行校验和校验

```

```

*/
    err = skb_copy_and_csum_datagram_iovec(skb,
                                              sizeof(struct udphdr),
                                              msg->msg_iov);
    if (err == -EINVAL)
        goto csum_copy_err;
}
/* 复制错误检查

```

```

*/
if (err)
    goto out_free;
/* 如果不是

```

peek动作，则增加相应的统计计数

```

*/
if (!peeked)
    UDP_INC_STATS_USER(sock_net(sk),
        UDP_MIB_INDATAGRAMS, is_udplite);
/* 更新套接字的最新的接收数据包时间戳及丢包消息

```

```

*/
sock_recv_ts_and_drops(msg, sk, skb);
/* 如果用户指定了保存对端地址的参数，则从数据包中复制地址和端口信息

```

```

*/
if (sin) {
    sin->sin_family = AF_INET;
    sin->sin_port = udp_hdr(skb)->source;
    sin->sin_addr.s_addr = ip_hdr(skb)->saddr;
    memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
}
/* 设置了接收控制消息

```

```

*/
if (inet->cmmsg_flags) {
    /* 接收控制消息如

```

TTL、

TOS等

```

*/
    ip_cmsg_recv(msg, skb);
}
/* 设置了已复制的字节长度

```

```

*/
err = copied;
if (flags & MSG_TRUNC)
    err = ulen;
out_free:
/* 释放接收到的这个数据包

```

```

*/
skb_free_datagram_locked(sk, skb);
out:
/* 返回读取的字节数

```

```

*/
return err;
/* 错误处理

```

```

*/

```

```
}

```

从上面的代码中，我们可以得到一个大部分书中都不会涉及的信息。先想一想，在读取一个UDP数据包时，如果传递给接口的缓存空间小于UDP数据包的实际大小时，结果会是什么样的呢？对于TCP来说，这个问题比较简单，因为它是流协议，没有数据报文边界，所以这次未读取的数据，会在下一次读取时被复制。但是UDP是基于数据包的，从上面的内核源码可以看到，当缓存小于UDP报文的实际大小时，内核会将报文截断，只复制缓存大小的数据，同时设置上MSG_TRUNC截断标志。这种情况，是很难从书本上了解到的，只有通过阅读源码才能理解其中的奥妙。

再进入__skb_recv_datagram，来查看UDP是如何接收报文的，代码如下：

```
struct sk_buff *__skb_recv_datagram(struct sock *sk, unsigned flags,
                                     int *peeked, int *err)
{
    struct sk_buff *skb;
    long timeo;
    /* 检查套接字是否出错

    */
    int error = sock_error(sk);
    if (error)
        goto no_packet;
    /* 得到超时时间，如果设置了

MSG_DONTWAIT，则超时为

0.

    */
    timeo = sock_rcvtimeo(sk, flags & MSG_DONTWAIT);
    do {
        unsigned long cpu_flags;
        spin_lock_irqsave(&sk->sk_receive_queue.lock, cpu_flags);
        /* 得到接收队列的第一个数据包

    */
        skb = skb_peek(&sk->sk_receive_queue);
        if (skb) {
            *peeked = skb->peeked;
            /* 如果只是查看动作，则要增加数据包的引用计数，并不用把数据包从队列中移除。

    */
            if (flags & MSG_PEEK) {
                skb->peeked = 1;
                atomic_inc(&skb->users);
            } else {
                /* 将数据包从接收队列中删除

    */

```

```

        __skb_unlink(skb, &sk->sk_receive_queue);
    }
}
spin_unlock_irqrestore(&sk->sk_receive_queue.lock, cpu_flags);
/* 得到了数据包, 直接返回

*/
if (skb)
    return skb;
/* 若已经没有了剩余的超时时间, 则跳转到

no_packet并返回

NULL */
error = -EAGAIN;
if (!timeo)
    goto no_packet;
/* 使

task在套接字上等待

*/
} while (!wait_for_packet(sk, err, &timeo));
return NULL;
no_packet:
*err = error;
return NULL;
}

```

如果当前的UDP套接字没有数据包, 则会进入wait_for_packet进行等待, 代码如下:

```

static int wait_for_packet(struct sock *sk, int *err, long *timeo_p)
{
    int error;
    /* 定义等待队列和回调的唤醒函数

*/
    DEFINE_WAIT_FUNC(wait, receiver_wake_function);
    /* 初始化等待队列, 需要注意的是

TASK_INTERRUPTIBLE。这表明进程在睡眠等待时, 是可以被中断的。

*/
    prepare_to_wait_exclusive(sk_sleep(sk), &wait, TASK_INTERRUPTIBLE);
    /* 检查套接字是否出错, 如被

RESET。如有错误, 则直接退出。

*/
    error = sock_error(sk);
    if (error)
        goto out_err;
    /* 若接收队列不为空, 则可以直接退出

*/

```

```

    if (!skb_queue_empty(&sk->sk_receive_queue))
        goto out;
    /* 检查套接字是否已经做了接收半关闭

*/
    if (sk->sk_shutdown & RCV_SHUTDOWN)
        goto out_noerr;
    /* 如果套接字是基于连接的，并且不是处于已连接状态或监听状态，则报错退出

*/
    error = -ENOTCONN;
    if (connection_based(sk) &&
        !(sk->sk_state == TCP_ESTABLISHED || sk->sk_state == TCP_LISTEN))
        goto out_err;
    /* 是否有未处理的信号

*/
    if (signal_pending(current))
        goto interrupted;
error = 0;
/* 将当前进程调度出去，直到超时，即进程已经休眠了设定的超时时间。但是由于某些原因，进程被提前唤
醒，所以需要保存返回的时间

*timeo_p, 表示还剩下多少时间。

*/
*timeo_p = schedule_timeout(*timeo_p);
out:
    finish_wait(sk_sleep(sk), &wait);
return error;
interrupted:
    error = sock_intr_errno(*timeo_p);
out_err:
    *err = error;
    goto out;
out_noerr:
    *err = 0;
    error = 1; goto out;
}

```

至此，UDP数据包的接收流程已经跟踪完毕。但是这里遗留了一个问题：在上面的代码中，报文是从接收队列中获得的，但是数据包又是如何被保存到套接字的接收队列中的呢？这个问题留到后面再做分解讨论。

14.4 TCP数据包的接收流程

14.3节跟踪了UDP数据包的接收流程，本节来分析一下TCP数据包的接收流程。根据TCP协议的复杂性可想而知，其接收流程自然也比UDP要繁琐得多。

```
int tcp_recvm(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
             size_t len, int nonblock, int flags, int *addr_len)
{
    struct tcp_sock *tp = tcp_sk(sk);
    int copied = 0;
    u32 peek_seq;
    u32 *seq;
    unsigned long used;
    int err;
    int target;          /* Read at least this many bytes */
    long timeo;
    struct task_struct *user_recv = NULL;
    int copied_early = 0;
    struct sk_buff *skb;
    u32 urg_hole = 0;
    /* 对套接字上锁

    */
    lock_sock(sk);
    err = -ENOTCONN;
    /* 如果套接字为监听状态，则跳转到退出分支

    */
    if (sk->sk_state == TCP_LISTEN)
        goto out;
    /* 与

    UDP类似，得到超时时间

    */
    timeo = sock_rcvtimeo(sk, nonblock);
    /* 设置了

    MSG_OOB标志，即带外数据，对于

    TCP来说，就是接收紧急数据

    */
    if (flags & MSG_OOB)
        goto rcv_urg;
    /* 得到与预读取

    TCP数据相对应的序列号

    */
    seq = &tp->copied_seq;
    if (flags & MSG_PEEK) {
        peek_seq = tp->copied_seq;
        seq = &peek_seq;
    }
    /*
    因为

    TCP是流协议，数据没有边界，所以需要计算接收数据的最小长度。

    1) 若设置了

    MSG_WAITALL，则目标为用户指定的长度；

    2) 不然，则选择套接字的低水线和用户指定长度的最小值；

    3) 如果第二种情况的最小值为

    0，则数据长度为

    1字节；

    */
    target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
    /*
    CONFIG_NET_DMA编译选项的含义为

    TCP接收复制卸载。

    利用
```

DMA来将接收到的数据复制到用户空间，从而节省

CPU.

```
*/
#ifdef CONFIG_NET_DMA
tp->ucopy.dma_chan = NULL;
preempt_disable();
skb = skb_peek_tail(&sk->sk_receive_queue);
{
    int available = 0;
    /* 接收队列中有未读的数据包

*/
    if (skb) {
        /* 计算可读的数据量

*/
        available = TCP_SKB_CB(skb)->seq + skb->len - (*seq);
    }
    if ((available < target) &&
        (len > sysctl_tcp_dma_copybreak) && !(flags & MSG_PEEK) &&
        !sysctl_tcp_low_latency &&
        dma_find_channel(DMA_MEMCPY)) {
        preempt_enable_no_resched();
        /* 确定
```

DMA要使用的数据段

```
*/
        tp->ucopy.pinned_list =
            dma_pin_iovec_pages(msg->msg_iov, len);
    } else {
        preempt_enable_no_resched();
    }
}
#endif
do {
    u32 offset;
    /* 判断是否正在读取紧急数据

*/
    if (tp->urg_data && tp->urg_seq == *seq) {
        /* 如果已经读取了一定量的数据，则结束读取

*/
        if (copied)
            break;
        /* 如果有未处理的信号，也结束读取

*/
        if (signal_pending(current)) {
            copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
            break;
        }
    }
    /* 遍历接收队列

*/
    skb_queue_walk(&sk->sk_receive_queue, skb) {
        /* Now that we have two receive queues this
         * shouldn't happen.
        */
        if (WARN(before(*seq, TCP_SKB_CB(skb)->seq),
            "recvmmsg bug: copied %X seq %X rcvnxt %X fl %X\n",
            *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt,
            flags))
            break;
        /* 取得在数据包中的偏移，即上次没有将这个数据包读取完毕

*/
        offset = *seq - TCP_SKB_CB(skb)->seq;
        /* syn标志会占用一个
```

sequence，所以偏移减一

```
*/
    if (tcp_hdr(skb)->syn)
        offset--;
    /* 若偏移小于数据包长度，则这个数据包就是要接收的数据包

*/
    if (offset < skb->len)
        goto found_ok_skb;
    /* 如果当前数据包包含
```

FIN标志，则跳转到

fin处

```
*/
    if (tcp_hdr(skb)->fin)
        goto found_fin_ok;
    WARN(!(flags & MSG_PEEK),
        "recvmmsg bug 2: copied %X seq %X rcvnxt %X fl %X\n",
```



```
        *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt, flags);
    }
    /* 若已经复制了超过目标的数据并且有积压的数据，则立刻跳出，并尝试处理积压数据。
```

```
*/
    if (copied >= target && !sk->sk_backlog.tail)
        break;
    /*
    这里针对是否已经复制了部分数据做了条件判断，而且每个分支中都有相似的条件判断，为什么要
```

分两种情况呢？因为在读取过程中，如果发生了同样的错误，只读取了部分数据，那么系统调用的

返回值要返回成功读取的字节数；而未读取任何数据，则返回

-1错误。

```
*/
if (copied) {
    /*
    已复制了部分数据，检查下面几个条件：
```

1) 套接字出错。

2) 连接已经关闭。

3) 套接字关闭了接收端。

4) 已经超时。

5) 有待处理的信号。

若有一个条件符合，则跳出接收数据循环。

```
*/
if (sk->sk_err ||
    sk->sk_state == TCP_CLOSE ||
    (sk->sk_shutdown & RCV_SHUTDOWN) ||
    !timeo ||
    signal_pending(current))
    break;
} else {
    /*
    若套接字设置了
```

SOCK_DONE标志，则跳出循环。

对于

TCP来说，被动关闭时，套接字会被设置上这个标志。这就意味着对端已经关闭，所以不

可能再有新的数据了。

```
*/
if (sock_flag(sk, SOCK_DONE))
    break;
/* 判断套接字是否出错
```

```
*/
if (sk->sk_err) {
    copied = sock_error(sk);
    break;
}
/* 套接字关闭了接收端
```

```
*/
if (sk->sk_shutdown & RCV_SHUTDOWN)
    break;
/* 套接字状态为关闭状态但又没有设置
```

SOCK_DONE标志，这种情况只发生在用户企图从一个未连接的套接字中读取数据时。

```
*/
if (sk->sk_state == TCP_CLOSE) {
```

```

        if (!sock_flag(sk, SOCK_DONE)) {
            copied = -ENOTCONN;
            break;
        }
        break;
    }
    /* 已经超时

```

```

*/
    if (!timeo) {
        copied = -EAGAIN;
        break;
    }
    /* 有未处理的信号

```

```

*/
    if (signal_pending(current)) {
        copied = sock_intr_errno(timeo);
        break;
    }
    /* 清除已经读取的数据包

```

```

*/
    tcp_cleanup_rbuf(sk, copied);
    /* 要进行低延时的

```

TCP处理

```

*/
    if (!sysctl_tcp_low_latency && tp->ucopy.task == user_rcv) {
        /* 保存用户进程地址

```

```

*/
    if (!user_rcv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
        user_rcv = current;
        tp->ucopy.task = user_rcv;
        tp->ucopy.iov = msg->msg_iov;
    }
    tp->ucopy.len = len;
    WARN_ON(tp->copied seq != tp->rcv_nxt &&
            T(flags & (MSG_PEEK | MSG_TRUNC)));
    /*
    处理完

```

receive queue. 需要处理

prequeue .

TCP套接字有三个队列，需要按照以下顺序来处理：

1)

receive_queue;
2)

prequeue:

3)

backlog.

```

*/
    if (!skb_queue_empty(&tp->ucopy.prequeue))
        goto do_prequeue;
    /* __ Set realtime policy in scheduler __ */
}
#ifdef CONFIG_NET_DMA
    if (tp->ucopy.dma_chan) {
        if (tp->rcv_wnd == 0 &&
            !skb_queue_empty(&sk->sk_async_wait_queue)) {
            /*
            接收窗口已经为

```

0. 并且有进程正在等待数据，这时就要尽快接收数据。所以这里的

dma 操作为同步的。

```

*/
    tcp_service_net_dma(sk, true);
    tcp_cleanup_rbuf(sk, copied);
} else
    dma_async_memcpy_issue_pending(tp->ucopy.dma_chan);
}
#endif
    if (copied >= target) {
        /* 若已经复制了超过目标的数据量，则释放该套接字

```

```

*/
    release_sock(sk);
    lock_sock(sk);
} else {
    /* 等待更多的数据

*/
    sk_wait_data(sk, &timeo);
}
#ifdef CONFIG_NET_DMA
tcp_service_net_dma(sk, false); /* Don't block */
tp->ucopy.wakeup = 0;
#endif
if (user_recv) {
    int chunk;
    /* Restore normal policy in scheduler */
    if ((chunk = len - tp->ucopy.len) != 0) {
        NET_ADD_STATS_USER(sock_net(sk),
            LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
        len -= chunk;
        copied += chunk;
    }
    /* 处理完

```

receive_queue. 再继续处理

```

prequeue */
    if (tp->rcv_nxt == tp->copied_seq &&
        !skb_queue_empty(&tp->ucopy.prequeue)) {
do_prequeue:
    tcp_prequeue_process(sk);
    /* 计算从

```

prequeue中读取的数据长度, 并调整相应的

len和

copied.

```

*/
    if ((chunk = len - tp->ucopy.len) != 0) {
        NET_ADD_STATS_USER(sock_net(sk),
            LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
        len -= chunk;
        copied += chunk;
    }
}
if ((flags & MSG_PEEK) &&
    (peek_seq - copied - urg_hole != tp->copied_seq)) {
    if (net_ratelimit())
        printk(KERN_DEBUG "TCP(%s:%d): Application bug, race in MSG_PEEK.\n",
            current->comm, task_pid_nr(current));
    peek_seq = tp->copied_seq;
}
continue;
found_ok_skb:
/* OK so how much can we use? */
/* 找到了正确的

```

skb, 计算该

skb未读的可用数据长度

```

*/
    used = skb->len - offset;
    /* 如果用户要读取的长度小于当前的剩余长度, 则调整可用长度

```

```

*/
    if (len < used)
        used = len;
    /* 判断是否有紧急数据。

```

TCP的紧急数据又称带外数据。在协议定义本身一直都有些争议。所以其实现代码也比较奇怪。一般

不推荐在日常编码中使用紧急数据

```

*/
    if (tp->urg_data) {
        /* 得到紧急数据的偏移

```

```

*/
    u32 urg_offset = tp->urg_seq - *seq;
    /* 判断紧急数据是否在我们读取的数据范围内

```

```

*/
    if (urg_offset < used) {
        if (!urg_offset) {
            /* 判断紧急数据是否在普通数据流中

```

```

*/
    if (!sock_flag(sk, SOCK_URGINLINE)) {
        /* 若不在普通数据流中, 则要忽略当前这个字节

```

```

*/
        ++*seq;
        urg_hole++;
        offset++;
        used--;
        if (!used)
            goto skip_copy;
    }
} else
    used = urg_offset;
}
}
/* 没有设置截断标志

```

```

*/
    if (!(flags & MSG_TRUNC)) {
        /* 先尝试使用

```

DMA来将数据复制到用户空间

```

*/
#ifdef CONFIG_NET_DMA
    if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
        tp->ucopy.dma_chan = dma_find_channel(DMA_MEMCPY);
    if (tp->ucopy.dma_chan) {
        tp->ucopy.dma_cookie = dma_skb_copy_datagram_iovec(
            tp->ucopy.dma_chan, skb, offset,
            msg->msg_iov, used,
            tp->ucopy.pinned_list);
        if (tp->ucopy.dma_cookie < 0) {
            printk(KERN_ALERT "dma_cookie < 0\n");
            /* Exception. Bailout! */
            if (!copied)
                copied = -EFAULT;
            break;
        }
        dma_async_memcpy_issue_pending(tp->ucopy.dma_chan);
        if ((offset + used) == skb->len)
            copied_early = 1;
    } else
#endif
    {
        /* 复制数据到用户空间

*/
        err = skb_copy_datagram_iovec(skb, offset,
            msg->msg_iov, used);
        if (err) {
            /* Exception. Bailout! */
            if (!copied)
                copied = -EFAULT;
            break;
        }
    }
}
/* 调整序列号

```

*seq. 已复制长度、剩余长度

```

*/
    *seq += used;
    copied += used;
    len -= used;
    /* 因为成功读取了数据，所以要调整

```

TCP套接字的接收缓存

```

*/
tcp_rcv_space_adjust(sk);
skip_copy:
/* 如果正在读取，并且已读取的序列号大于紧急数据，则意味着已经读取完了紧急数据，那么就

```

要重置

urg_data. 并且进行

TCP快速路径检查（如果通过了检查条件，则打开快速路径开关。

打开快速路径的时候，表示接收的数据包是预期的数据包。

TCP接收数据包时会做比较少的检

查，因此接收更为快速）

```

*/
    if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
        tp->urg_data = 0;
        tcp_fast_path_check(sk);
    }
/* 使用的数据长度加上偏移若小于数据包的长度，则该数据包可以继续使用

```

```

*/
    if (used + offset < skb->len)
        continue;
/* 如果该数据包有

```

FIN标志, 则跳转到

```
found_fin_ok */
    if (tcp_hdr(skb)->fin)
        goto found_fin_ok;
    /* 如果没有设置
```

MSG_PEEK标志, 则需要从接收队列中消耗掉这个数据包, 并根据

```
copied_
    early标志, 将其直接释放, 或者放置到异步队列
```

```
*/
    if (!(flags & MSG_PEEK)) {
        sk_eat_skb(sk, skb, copied_early);
        copied_early = 0;
    }
    continue;
found_fin_ok:
    /* 这里开始处理
```

FIN数据包

```
*/
    /* FIN标志也占用一个序列号, 因此要给序列号加一
```

```
*/
    ++*seq;
    /* 与上文相同, 不再重复注释
```

```
*/
    if (!(flags & MSG_PEEK)) {
        sk_eat_skb(sk, skb, copied_early);
        copied_early = 0;
    }
    /* 接收到
```

FIN标志, 表示对端已经关闭了写通道, 那么对于本端来说, 这是最后一个可读数据包。

因此退出循环

```
*/
    break;
} while (len > 0);
if (user_recv) {
    /* prequeue队列中仍然有未读取的数据包
```

```
*/
    if (!skb_queue_empty(&tp->ucopy.prequeue)) {
        int chunk;
        /* 设置要读取的长度
```

```
*/
    tp->ucopy.len = copied > 0 ? len : 0;
    /* 处理
```

prequeue队列

```
*/
    tcp_prequeue_process(sk);
    if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
        NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
        len -= chunk;
        copied += chunk;
    }
    tp->ucopy.task = NULL;
    tp->ucopy.len = 0;
}
#ifdef CONFIG_NET_DMA
tcp_service_net_dma(sk, true); /* Wait for queue to drain */
tp->ucopy.dma_chan = NULL;
if (tp->ucopy.pinned_list) {
    dma_unpin_iovec_pages(tp->ucopy.pinned_list);
    tp->ucopy.pinned_list = NULL;
}
#endif
/* 释放已经读取的数据包
```

```
*/
tcp_cleanup_rbuf(sk, copied);
/* 释放套接字控制权
```

```
*/
release_sock(sk);
return copied;
out:
release_sock(sk);
return err;
recv_urg:
/* 接收紧急数据
```

```
*/
    err = tcp_recv_urg(sk, msg, len, flags);
    goto out;
}
```

尽管上面的tcp_recvmsg已经加了大量的注释，但是由于这个函数的逻辑过于复杂，再加上TCP接收队列的多样性，即使已经看完了这个函数的实现，却仍然无法清楚地掌握它的整体脉络。接下来，我们

14.5 TCP套接字的三个接收队列

在Linux内核中，除了错误队列外，TCP套接字一共有三个接收队列。它们分别是struct sock中的sk_receive_queue和sk_backlog，以及struct tcp_sock中的prequeue。先简单介绍一下它们各自的用途，然后再看具体的代码实现。sk_receive_queue是真正的接收队列，收到的TCP数据包经过检查和处理后，就会保存在这个队列中，用户态也是从这里读取数据的。sk_backlog是socket正处于用户进程上下文（即用户正在对socket进行系统调用，如recv），当Linux内核收到数据包时，在软中断的处理过程中，内核会将数据包保存在sk_backlog中，然后直接返回。而prequeue则是在该socket没有正在被用户进程使用时，由软中断直接将数据包保存在prequeue中，并返回。从上面的说明可以看出，对于TCP套接字，它不管用户态是否正在使用套接字，都不做真正的处理，而是把数据包保存在队列中，这是为什么呢？这是因为TCP协议相对复杂，内核为了尽快让软中断结束，就不进行多余的处理了，尽量在用户进程上下文中处理数据包。下面来看看TCP相关的源代码。

首先，查看TCP的接收处理函数tcp_v4_rcv中的一部分代码：

```
bh_lock_sock_nested(sk);
ret = 0;
if (!sock_owned_by_user(sk)) {
    /* 用户态没有正在使用这个套接字

*/
#ifdef CONFIG_NET_DMA
    struct tcp_sock *tp = tcp_sk(sk);
    if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
        tp->ucopy.dma_chan = dma_find_channel(DMA_MEMCPY);
    if (tp->ucopy.dma_chan)
        ret = tcp_v4_do_rcv(sk, skb);
    else
#endif
    {
        /* 先尝试保存到
```

prequeue中，若失败的话再进入

TCP真正的处理函数中

```
*/
        if (!tcp_prequeue(sk, skb))
            ret = tcp_v4_do_rcv(sk, skb);
    }
    /* 若该套接字正在被用户态使用，则将数据包保存到
```

backlog中。如果失败的话，就丢弃这个包。

```
*/
    } else if (unlikely(sk_add_backlog(sk, skb))) {
        bh_unlock_sock(sk);
        NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
        goto discard_and_relse;
```

```
}  
bh_unlock_sock(sk);
```

然后进入tcp_prequeue，看看什么时候会返回失败，代码如下：

```
static inline int tcp_prequeue(struct sock *sk, struct sk_buff *skb)  
{  
    struct tcp_sock *tp = tcp_sk(sk);  
    /* 配置了低延时
```

TCP，或者该套接字没有对应的用户态进程，返回失败。让内核直接处理

TCP数据包。

```
*/  
    if (sysctl_tcp_low_latency || !tp->ucopy.task)  
        return 0;  
    /* 将数据包追加到
```

prequeue队列中，并增加相应的内存统计。

```
*/  
    __skb_queue_tail(&tp->ucopy.prequeue, skb);  
    tp->ucopy.memory += skb->truesize;  
    if (tp->ucopy.memory > sk->sk_rcvbuf) {  
        /* 当超过了套接字指定的接收缓存大小时
```

```
*/  
    struct sk_buff *skb1;  
    BUG_ON(sock_owned_by_user(sk));  
    /* 将数据包从
```

prequeue中转移到

backlog中

```
*/  
    while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL) {  
        sk_backlog_rcv(sk, skb1);  
        NET_INC_STATS_BH(sock_net(sk),  
            LINUX_MIB_TCPPREQUEUEDROPPED);  
    }  
    tp->ucopy.memory = 0;  
} else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {  
    /* 如果该数据包是
```

prequeue中的第一个数据包，则唤醒在该套接字中等待接收的进程

```
*/  
    wake_up_interruptible_sync_poll(sk_sleep(sk),  
        POLLIN | POLLRDNORM | POLLRDBAND);  
    /* 如果
```


ack定时器没有被调度，则设置

ack定时器

```
*/
if (!inet_csk_ack_scheduled(sk))
    inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
        (3 * tcp_rto_min(sk)) / 4,
        TCP_RTO_MAX);
}
return 1;
}
```

然后查看sk_add_backlog，代码如下：

```
static inline __must_check int sk_add_backlog(struct sock *sk, struct sk_buff *skb)
{
    /* 接收队列已满，则返回
```

ENOBUFFS错误。所谓的接收队列已满，即接收缓存的数据包占用的内存超过了限制。

```
*/
if (sk_rcvqueues_full(sk, skb))
    return -ENOBUFFS;
/* 将数据包追加到
```

backlog队列中，并增加相应的内存统计。

```
*/
sk_add_backlog(sk, skb);
sk->sk_backlog.len += skb->truesize;
return 0;
}
```

看完这些代码后，我们应该产生一个疑问。既然prequeue和backlog都是保存的未经处理的TCP数据包，那么为什么还需要两个不同的队列呢？为了解答这个疑问，就需要研究内核是如何使用这两个队列的了。前面的代码是这两个队列的写入操作，接下来我们看一下这两个队列是何时被读取的。

prequeue队列的处理函数是tcp_prequeue_process，它是在TCP的读取数据函数tcp_recvmsg中被调用的。在tcp_recvmsg的入口，内核会调用lock_sock来设置sk->sk_lock.owned，表示该套接字由用户进程所占有，然后会对receive_queue和prequeue中的数据包进行处理。正因为sock被用户进程占用时，会访问prequeue队列，所以为了避免竞争，软中断在收到数据包时就只能把数据包保存到backlog中。那么为什么当sock不被用户进程占用时，软中断不将数据包保存到backlog中，而是保存到prequeue中呢？

要回答这个问题，还是要继续查看backlog是何时被读取的。让人觉得有点出乎意料的是，backlog的数据包居然是在__release_sock中被处理的。

```
static void __release_sock(struct sock *sk)
{
    __release(&sk->sk_lock.slock)
    __acquires(&sk->sk_lock.slock)
    {
        struct sk_buff *skb = sk->sk_backlog.head;
        /* 处理
```

backlog队列的数据包

```
*/
do {
    sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
    bh_unlock_sock(sk);
    do {
        struct sk_buff *next = skb->next;
        WARN_ON_ONCE(skb_dst_is_noref(skb));
        skb->next = NULL;
        sk_backlog_rcv(sk, skb);
        /*
         * We are in process context here with softirqs
         * disabled, use cond_resched_softirq() to preempt.
         * This is safe to do because we've taken the backlog
         * queue private:
         */
        cond_resched_softirq();
        skb = next;
    } while (skb != NULL);
    bh_lock_sock(sk);
} while ((skb = sk->sk_backlog.head) != NULL);
/*
 * Doing the zeroing here guarantee we can not loop forever
 * while a wild producer attempts to flood us.
 */
sk->sk_backlog.len = 0;
}
```

不过这也解释了对于TCP套接字，为什么需要两个队列来保存未处理的数据包。

对于套接字的使用情况，一共有两个状态：

- 用户进程正在占用该套接字。
- 用户进程未占用该套接字。

而内核在任何情况下，都要尽量保证尽快返回软中断，以避免资源竞争。因此，在套接字的这两个状态下，都要保证软中断可以毫无阻塞地将数据包保存到未处理队列中，自然也就需要两个队列了。当用户进程正在占用套接字时，其会访问prequeue，那么软中断就将数据包保存到backlog中。当用户放弃对套接字的占用时，其会访问backlog，而这时，软中断就会将数据包保存到prequeue中。

14.6 从网卡到套接字

对于一般的套接字编程来说，大多是应用编程，所以基本上都是UDP或TCP协议的套接字。前面两章是从应用层次的角度，自上而下地分析了UDP和TCP数据包的发送和接收流程。但同时也有了一个新的问题，数据包是如何进入对应套接字的接收缓冲区的呢？本章将从网卡接收到数据包开始，一直跟踪到内核将数据包放入到对应的套接字缓冲区中为止。

14.6.1 从硬中断到软中断

对于网卡来说，数据包的到达是一个无法预料的事件，系统需要通过某种手段来得知该事件。一般来说，有两种方式：轮询和中断。用直白的语言来描述，轮询就是CPU不断地问网卡：“你那有准备好的数据包吗？”如果网卡回答有数据包的话，CPU就进行处理，不然要么干点别的，要么继续问。中断则是没有数据包时，CPU该干嘛干嘛，若网卡收到数据包，就直接喊话“喂，有活干了”。于是CPU赶紧把手头的工作保存一下，并尽快响应任务。第一种方式，毫无疑问会造成CPU的浪费。因为在网卡没有数据包的时候，CPU还要浪费计算周期来询问网卡。第二种中断方式看上去很美，网卡没有数据的时候，CPU可以做其他的事情；有数据的时候，就可以及时处理。然而在实际应用中，中断方式也有很大的问题。在CPU响应中断时，为了不影响当前的工作，需要将当前工作的上下文保存起来，然后再进行中断处理。试想，当前千兆、万兆网卡已经非常普遍，若是那时网卡满负载，那么每秒钟就会产生大量的中断。除了切换过程带来的计算代价，上下文的切换还会导致CPU Cache的失效——这对高性能设备来说，是一个不可忽视的问题。于是，Linux对这两种方式进行了折中，引入了一个New API，缩写为NAPI。简单来说，在CPU响应网卡中断时，不再仅仅是处理一个数据包就退出，而是使用轮询的方式继续尝试处理新数据包，直到没有新数据包到来，或者达到设置的一次中断最多处理的数据包个数。这个NAPI同时兼有了轮询和中断两种方式的特点。

网卡硬中断的处理是在网卡驱动中进行的，这个与硬件的联系过于紧密，我们可以忽略细节。只需要知道对于支持NAPI的网卡来说，其读取数据包的硬中断处理函数会调用__napi_schedule将网卡加入NAPI的poll list中，代码如下：

```
void __napi_schedule(struct napi_struct *n)
{
    unsigned long flags;
    /* 禁止本地中断，保护添加

poll list的临界区

*/
    local_irq_save(flags);
    /* 加入到当前

CPU的

poll列表中

*/
    napi_schedule(&_get_cpu_var(softnet_data), n);
    local_irq_restore(flags);
}
```

进入____napi_schedule，代码如下：

```
static inline void ____napi_schedule(struct softnet_data *sd,
                                   struct napi_struct *napi)
{
    /* 将

napi加入队尾

*/
    list_add_tail(&napi->poll_list, &sd->poll_list);
    /* 触发当前

CPU接收软中断（实际上是设置一个标志位）

*/
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

关于中断处理为什么要分为硬中断和软中断（也经常被称为上下部分）的解释已经很多了。简单地说：硬中断处理是一个特殊的上下文，CPU会屏蔽掉绝大部分中断，并且有不少的限制。所以硬中断应尽可能快地处理，以提高系统的响应速度，因此内核将具体的处理工作放到了软中断中。

14.6.2 软中断处理

14.6.1节中，我们看到硬中断通过设置标志位“触发”了软中断。那么内核又是何时处理软中断的呢？目前，在以下几种条件下，内核会检查是否需要处理软中断：

- 退出硬中断上下文时。
- 重新enable软中断时。
- 每个CPU都有一个ksoftirqd的内核线程。当内核的软中断数量过多时，就会唤醒该线程循环处理软中断。

接收数据包的软中断处理函数为net_rx_action，代码如下：

```
static void net_rx_action(struct softirq_action *h)
{
    /* 接收数据包的

per cpu队列

*/
    struct softnet_data *sd = &__get_cpu_var(softnet_data);
    /* 最长的运行时间限制为

2个

jiffies */
    unsigned long time_limit = jiffies + 2;
    /*
    一次软中断最多处理的包个数。

netdev_budget的值为

/proc/sys/net/core/netdev_budget
*/
    int budget = netdev_budget;
    void *have;
    /* 因为网卡驱动会访问

poll list, 因此需要禁止本地硬中断以进行保护

*/
    local_irq_disable();
    /* 遍历加入到

poll链表的所有网卡
```

```

*/
while (!list_empty(&sd->poll_list)) {
    struct napi_struct *n;
    int work, weight;
    /* 如果已经处理完了允许的最大包个数，或超出了允许的时间限制，则退出此次处理

*/
    if (unlikely(budget <= 0 || time_after(jiffies, time_limit))) {
        /* 这时退出此次收包中断只是为了避免过长时间地占用

```

CPU，所以跳到

softnet_break，再触发一次收包软中断，以便下次继续处理

```

*/
    goto softnet_break;
}
/* 打开本地硬中断

```

```

*/
local_irq_enable();
/* 这里在打开硬中断时，虽然访问了

```

poll_list，但仍然是安全的。因为硬中断只是在往

poll_list的末尾插入，并不会影响第一个元素。

```

*/
n = list_first_entry(&sd->poll_list, struct napi_struct, poll_list);
/* 获得该设备的

```

netpoll锁

```

*/
have = netpoll_poll_lock(n);
/* 得到该网卡的权重，其意义一般为在这个网卡上接收几个数据包

```

```

*/
weight = n->weight;
/* This NAPI_STATE_SCHED test is for avoiding a race
 * with netpoll's poll_napi(). Only the entity which
 * obtains the lock and sees NAPI_STATE_SCHED set will
 * actually make the ->poll() call. Therefore we avoid
 * accidentally calling ->poll() when NAPI is not scheduled.
 */
work = 0;
/* 再次检查该网卡是否有

```

NAPI调用（因为与

netpoll有竞争）

```

*/
    if (test_bit(NAPI_STATE_SCHED, &n->state)) {
        /* 对网卡进行查询操作,

```

work值为读取的数据包个数

```

*/
        work = n->poll(n, weight);
        trace_napi_poll(n);
    }
    WARN_ON_ONCE(work > weight);
    /* 更新包预算即目前还可以读取的数据包个数

```

```

*/
    budget -= work;
    local_irq_disable();
    /* 判断从该网卡读取的数据包是否达到预算个数

```

```

*/
    if (unlikely(work == weight)) {
        /* 判断该网卡的

```

NAPI是否被禁止了

```

*/
        if (unlikely(napi_disable_pending(n))) {
            /* 若

```

NAPI已经被禁止了，则执行

NAPI的完成处理

```

*/
        local_irq_enable();
        napi_complete(n);
        local_irq_disable();
    } else {
        /* 若该设备仍要继续进行

```

NAPI操作，则将其移至队尾

```

*/
        list_move_tail(&n->poll_list, &sd->poll_list);
    }
    /* 释放

```

netpoll锁

```

*/
    netpoll_poll_unlock(have);
}
out:
    /* 执行

```


RPS处理并打开本地硬中断

```
*/
net_rps_action_and_irq_enable(sd);
#ifdef CONFIG_NET_DMA
/* 启动未处理的
```

DMA操作

```
*/
dma_issue_pending_all();
#endif
return;
softnet_break:
/* 本次没有接收完所有的数据包，再触发一次软中断

*/
sd->time_squeeze++;
raise_softirq_irqoff(NET_RX_SOFTIRQ);
goto out;
}
```

在这个收包软中断处理函数中，CPU会遍历poll列表，调用挂载到NAPI列表上的网卡回调函数poll，来轮询接收数据包。所以，我们还需要通过驱动代码，来跟踪收包流程。在此，以Intel的e1000网卡驱动为例来进行讲解，在使用NAPI的情况下，其收包流程为

net_rx_action→e1000_clean→e1000_clean_rx_irq→e1000_receive_skb→napi_gro_receive→netif_receive_skb。在这个调用链上，有一个函数napi_gro_receive，其用来支持GRO（Generic Receive Offload）。这个GRO则是用于减轻CPU的处理压力的。大家可以计算一下，对于10GB、100GB的网卡来说，即使每个数据包都是1500字节（以太网的最大MTU，暂不考虑Jumbo帧），那么每秒钟系统需要处理多少个数据包？因此，为了减轻CPU的负担，Linux内核在驱动层引入了GRO，它会将符合条件的数据包合并为一个数据包再传递给系统协议栈。在此，我们只关注数据包的接收流程，就不研究GRO的实现了。有兴趣的读者可以自行阅读源代码。

14.6.3 传递给协议栈流程

数据包在脱离驱动层后，就进入了netif_receive_skb，代码如下：

```
int netif_receive_skb(struct sk_buff *skb)
{
    /* 判断是否在入队前给数据包打时间戳

    */
    if (netdev_tstamp_prequeue)
        net_tstamp_check(skb);
    if (skb_defer_rx_timestamp(skb))
        return NET_RX_SUCCESS;
    /* 是否打开了

RPS (
Receive Packet Steering) 编译开关，其根据数据包的

IP地址和端口号进

行

hash运算，将其发送给对应的

CPU。这样，一方面保证了

CPU间的负载均衡，另一方面将同一特征

的数据包发给相同的

CPU，可以提高

cache的命中率。

*/
#ifdef CONFIG_RPS
{
    struct rps_dev_flow voidflow, *rflow = &voidflow;
    int cpu, ret;
    rcu_read_lock();
    /* 根据
```

RPS算法，计算得到处理这个数据包的

```
CPU */
    cpu = get_rps_cpu(skb->dev, skb, &rflow);
    /* 当
```

CPU大于等于

0时，表示

RPS计算得到了正确的

```
CPU */
    if (cpu >= 0) {
        /* 向其他
```

CPU的接收队列追加这个数据包

```
*/
        ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);
        rcu_read_unlock();
    } else {
        /* 由本
```

CPU处理该数据包

```
*/
        rcu_read_unlock();
        ret = __netif_receive_skb(skb);
    }
    return ret;
}
#else
/* 本
```

CPU继续处理该数据包

```
*/
    return __netif_receive_skb(skb);
#endif
}
```

这里可以看出netif_receive_skb只是对__netif_receive_skb的封装，增加了对RPS的支持。继续跟进__netif_receive_skb，代码如下：

```
static int __netif_receive_skb(struct sk_buff *skb)
{
    struct packet_type *ptype, *pt_prev;
    rx_handler_func_t *rx_handler;
    struct net_device *orig_dev;
    struct net_device *null_or_dev;
    bool deliver_exact = false;
    int ret = NET_RX_DROP;
    bel6 type;
    /* 如果没有打开入队前采样数据包的时间戳功能，则需要在这里进行数据包时间戳采样
```

```

*/
if (!netdev_tstamp_prequeue)
    net_timestamp_check(skb);
trace_netif_receive_skb(skb);
/* 判断是否由

```

netpoll处理

```

*/
if (netpoll_receive_skb(skb))
    return NET_RX_DROP;
/* 设置网卡的入口网卡

*/
if (!skb->skb_iif)
    skb->skb_iif = skb->dev->ifindex;
orig_dev = skb->dev;
/* 初始化数据包的网络层首部、传输层首部，以及二层

```

MAC首部的长度

```

*/
skb_reset_network_header(skb);
skb_reset_transport_header(skb);
skb_reset_mac_len(skb);
pt_prev = NULL;
rcu_read_lock();
another_round:
    this_cpu_inc(softnet_data.processed);
/* 如果是

```

802.1Q协议的数据包

```

*/
if (skb->protocol == cpu_to_be16(ETH_P_8021Q)) {
    /* 则去掉

```

```

vlan tag */
    skb = vlan_untag(skb);
    if (unlikely(!skb))
        goto out;
}
/* 内核打开了包分类编译选项

```

```

*/
#ifdef CONFIG_NET_CLS_ACT
/* 如果数据包被设置了流控结果，则跳过后面的流控处理

```

```

*/
if (skb->tc_verd & TC_NCLS) {
    skb->tc_verd = CLR_TC_NCLS(skb->tc_verd);
    goto ncls;
}
#endif
/* 遍历注册在

```

ptype_all上的所有节点。

ptype_all上的节点需要处理收到的所有以太网数据包

```
*/
list_for_each_entry_rcu(ptype, &ptype_all, list) {
    /* 如果注册节点没有绑定网卡，或者绑定的网卡与数据包接收的网卡相同，则这个节点符合接收数据包的条件

*/
    if (!ptype->dev || ptype->dev == skb->dev) {
        /* 将数据包传递给对应的处理函数

*/
        if (pt_prev)
            ret = deliver_skb(skb, pt_prev, orig_dev);
        pt_prev = ptype;
    }
}
/* 内核打开了包分类编译选项

*/
#ifdef CONFIG_NET_CLS_ACT
    skb = handle_ing(skb, &pt_prev, &ret, orig_dev);
    if (!skb)
        goto out;
ncls:
#endif
/* 如果这个数据包带有
```

vlan标签

```
*/
if (vlan_tx_tag_present(skb)) {
    if (pt_prev) {
        /* 则将数据包传递给之前确定的上层协议

*/
        ret = deliver_skb(skb, pt_prev, orig_dev);
        pt_prev = NULL;
    }
    /* 进行
```

vlan的处理

```
*/
if (vlan_do_receive(&skb))
    goto another_round;
else if (unlikely(!skb))
    goto out;
}
/*
判断该设备是否注册了接收处理函数。
```

设备上何时会注册接收处理函数呢？

netdev_rx_handler_register是注册设备接收处理函数的接

口。通过搜索

netdev_rx_handler_register的调用者，可以发现当网卡作为

bond加入桥接，或者

创建

macvlan时，会注册网卡的处理函数。使用这种方式，就做到了网卡接收处理函数与接收框架的解

耦。对于框架来说，通过这个回调函数（用函数指针实现的，内核中充斥着这样的代码），可以完全不用

了解具体的细节。未来增加更多的网卡处理函数时，只需要在该具体实现上，调用注册函数，而不用更改

接收框架的代码。

```
*/
rx_handler = rcu_dereference(skb->dev->rx_handler);
if (rx_handler) {
    if (pt_prev) {
        /* 将数据包传递给之前确定的上层协议

*/
        ret = deliver_skb(skb, pt_prev, orig_dev);
        pt_prev = NULL;
    }
    /* 调用在设备上注册的处理函数

*/
    switch (rx_handler(&skb)) {
case RX_HANDLER_CONSUMED:
    /* 处理函数已经消耗了这个数据包，直接跳至退出

*/
        ret = NET_RX_SUCCESS;
        goto out;
case RX_HANDLER_ANOTHER:
    /* 跳至
```

another_round，即跳至函数开头，重新处理

```
*/
        goto another_round;
case RX_HANDLER_EXACT:
```

```
/* 指示必须严格匹配接收网卡
```

```
*/
    deliver_exact = true;
case RX_HANDLER_PASS:
    /* 继续后面的处理

*/
    break;
default:
    BUG();
}
/* 如果数据包还带有
```

vlan tag, 则证明该数据包是发给其他终端的

```
*/
if (vlan_tx_nonzero_tag_present(skb))
    skb->pkt_type = PACKET_OTHERHOST;
/* deliver only exact match when indicated */
null_or_dev = deliver_exact ? skb->dev : NULL;
/* 根据数据包的类型, 遍历对应的处理函数

*/
type = skb->protocol;
list_for_each_entry_rcu(ptype,
    &ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
    /* 如果数据包类型匹配, 并且接收接口设备也匹配, 则证明这是正确的协议处理函数。然后调用前面的响应处理函数, 将数据包传递给上层协议

*/
    if (ptype->type == type &&
        (ptype->dev == null_or_dev || ptype->dev == skb->dev ||
         ptype->dev == orig_dev)) {
        if (pt_prev)
            ret = deliver_skb(skb, pt_prev, orig_dev);
        pt_prev = ptype;
    }
}
/* 最后检查
```

pt_prev是否为真。若为真, 则表示前面有匹配的处理函数, 然后进行调用。如果为假, 则

表示对于这个数据包, 内核没有对应的处理函数, 那就直接释放这个数据包。

```
*/
if (pt_prev) {
    ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
} else {
    atomic_long_inc(&skb->dev->rx_dropped);
    kfree_skb(skb);
    /* Jamal, now you will not able to escape explaining
     * me how you were going to use this. :-)
     */
    ret = NET_RX_DROP;
}
out:
    rcu_read_unlock();
    return ret;
}
```

14.6.4 IP协议处理流程

以IPv4的协议栈处理为例进行讲解，首先来看看IPv4协议是如何注册处理回调函数的。在inet_init中，调用了dev_add_pack（&ip_packet_type）进行了IPv4协议的注册。ip_packet_type的定义为：

```
static struct packet_type ip_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = ip_rcv,
    .gso_send_check = inet_gso_send_check,
    .gso_segment = inet_gso_segment,
    .gro_receive = inet_gro_receive,
    .gro_complete = inet_gro_complete,
};
```

因此，ip_rcv为IPv4协议数据包的入口函数。下面来看看该函数：

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
{
    const struct iphdr *iph;
    u32 len;
    /*
     * 如果该数据是发给其他终端的，则丢弃这个数据包。
```

这里的

pkt_type是根据二层地址来判断是否发给本机的。

```
*/
if (skb->pkt_type == PACKET_OTHERHOST)
    goto drop;
IP_UPD_PO_STATS_BH(dev_net(dev), IPSTATS_MIB_IN, skb->len);
/* 对数据包进行共享检查，保证
```

IP协议处理的数据包独享一个

skb。

```
*/
if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
    IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
    goto out;
}
/* 检查
```

IP首部，如果数据包小于

IP首部的大小，则出错

```
*/
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
```



```
        goto inhdr_error;
/* 得到
```

IP首部地址

```
*/
iph = ip_hdr(skb);
/* 根据
```

RFC标准，

IP首部小于

20字节，或者版本号不是

4的，就报错丢弃

```
*/
if (iph->ihl < 5 || iph->version != 4)
    goto inhdr_error;
/* 根据
```

IP首部指定的长度，再次检查数据包的大小

```
*/
if (!pskb_may_pull(skb, iph->ihl*4))
    goto inhdr_error;
/* 重新获取
```

IP首部地址。之所以要重新获取，是因为

pskb_may_pull可能要重新申请

```
skb */
iph = ip_hdr(skb);
/* 校验
```

IPv4的校验和

```
*/
if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
    goto inhdr_error;
/* 对数据包长度进行检查
```

```
*/
len = ntohs(iph->tot_len);
if (skb->len < len) {
    IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INTRUNCATEDPKTS);
```

```

        goto drop;
    } else if (len < (iph->ihl*4))
        goto inhdr_error;
    /* Our transport medium may have padded the buffer out. Now we know it
     * is IP we can trim to the true length of the frame.
     * Note this now means skb->len holds ntohs(iph->tot_len).
     */
    /* 因为传输媒介可能会给数据包进行补齐，现在已经根据

```

IP首部明确了数据包的长度，因此需要将数据

包的长度变为真正的长度并改变

```

*/
if (pskb_trim_rcsum(skb, len)) {
    IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
    goto drop;
}
/* 重置数据包的控制块信息

```

```

*/
memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));
/* 重置数据包的套接字信息

```

```

*/
skb_orphan(skb);
/* 遍历执行

```

netfilter在

PREROUTING点上的规则，如果数据包没有被丢弃，则进入

```

ip_rcv_finish */
return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,
    ip_rcv_finish);
inhdr_error:
    IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
drop:
    kfree_skb(skb);
out:
    return NET_RX_DROP;
}

```

本文不分析netfilter的相关代码。数据包经过netfilter的PREROUTING处的规则后，进入了ip_rcv_finish，代码如下：

```

static int ip_rcv_finish(struct sk_buff *skb)
{
    const struct iphdr *iph = ip_hdr(skb);
    struct rtable *rt;
    /* 如果数据包没有设置路由信息，则进行路由查询

    */
    if (skb_dst(skb) == NULL) {
        int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
            iph->tos, skb->dev);
        if (unlikely(err)) {
            /* 若查找路由失败，增加相应的错误计数，并丢弃数据包

```

```

*/
        if (err == -EHOSTUNREACH)
            IP_INC_STATS_BH(dev_net(skb->dev),
                            IPSTATS_MIB_INADDRERRORS);
        else if (err == -ENETUNREACH)
            IP_INC_STATS_BH(dev_net(skb->dev),
                            IPSTATS_MIB_INNOROUTES);
        else if (err == -EXDEV)
            NET_INC_STATS_BH(dev_net(skb->dev),
                            LINUX_MIB_IPRPFILTER);
        goto drop;
    }
}
#ifdef CONFIG_IP_ROUTE_CLASSID
if (unlikely(skb_dst(skb)->tclassid)) {
    struct ip_rt_acct *st = this_cpu_ptr(ip_rt_acct);
    u32 idx = skb_dst(skb)->tclassid;
    st[idx&0xFF].o_packets++;
    st[idx&0xFF].o_bytes += skb->len;
    st[(idx>>16)&0xFF].i_packets++;
    st[(idx>>16)&0xFF].i_bytes += skb->len;
}
#endif
/* iph大于

```

5, 即首部长度大于固定首部长度

20字节。因此说明该

IP报文具有

IP选项, 于是调用

ip_rcv_options处理

IP选项

```

*/
if (iph->ihl > 5 && ip_rcv_options(skb))
    goto drop;
/* 根据路由类型, 增加相应的计数

*/
rt = skb_rtable(skb);
if (rt->rt_type == RTN_MULTICAST) {
    IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,
                        skb->len);
} else if (rt->rt_type == RTN_BROADCAST)
    IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,
                        skb->len);
/* 调用路由的输入函数

*/
return dst_input(skb);
drop:
    kfree_skb(skb);
    return NET_RX_DROP;
}

```

对于发往本机的数据包来说, 其路由输入函数为ip_local_deliver, 代码如下:

```
int ip_local_deliver(struct sk_buff *skb)
{
    /* 该数据包是一个
```

IP分片数据包

```
*/
    if (ip_is_fragment(ip_hdr(skb))) {
        /* 进行
```

IP分片重组处理

```
*/
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
            return 0;
    }
    /* 遍历执行
```

netfilter在

LOCAL_IN上的规则，如为丢弃，则进入

```
ip_local_deliver_finish */
    return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
        ip_local_deliver_finish);
}
```

进入ip_local_deliver_finish，代码如下：

```
static int ip_local_deliver_finish(struct sk_buff *skb)
{
    struct net *net = dev_net(skb->dev);
    /* 拉出
```

IP报文首部，因为马上就要脱离

IP层，进入传输层了。

```
*/
    skb_pull(skb, ip_hdrlen(skb));
    /* 设置传输层首部地址
```

```
*/
    skb_reset_transport_header(skb);
    rcu_read_lock();
    {
        /* 得到传输层协议
```

```
*/
        int protocol = ip_hdr(skb)->protocol;
        int hash, raw;
```

```

const struct net_protocol *ipprot;
resubmit:
/* 将数据包传递给对应的原始套接字

*/
raw = raw_local_deliver(skb, protocol);
/* 根据传输协议确定对应的

```

inet协议

```

*/
hash = protocol & (MAX_INET_PROTOS - 1);
ipprot = rcu_dereference(inet_protos[hash]);
if (ipprot != NULL) {
/* 找到了匹配传输层的协议

*/
int ret;
/* 检查名称空间是否匹配

*/
if (!net_eq(net, &init_net) && !ipprot->netns_ok) {
if (net_ratelimit())
printk("%s: proto %d isn't netns-ready\n",
func_, protocol);
kfree_skb(skb);
goto out;
}
/* 协议的安全策略检查

*/
if (!ipprot->no_policy) {
if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
kfree_skb(skb);
goto out;
}
nf_reset(skb);
}
/* 将数据包传递给传输层处理

*/
ret = ipprot->handler(skb);
if (ret < 0) {
protocol = -ret;
goto resubmit;
}
IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
} else {
/* 没有对应的传输层协议

*/
if (!raw) {
/* 若没有匹配的原始套接字，则进行安全策略检查

*/
if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
/* 若没有对应的安全策略，则使用

```

ICMP返回不可达错误

```
*/
        IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);
        icmp_send(skb, ICMP_DEST_UNREACH,
                   ICMP_PROT_UNREACH, 0);
    }
    } else
        IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
    kfree_skb(skb);
}
}
out:
    rcu_read_unlock();
    return 0;
}
```

14.6.5 大师的错误？原始套接字的接收

在UNP1的28.4“Raw Socket Input”一节中，Stevens大师是这样说的：

Received UDP packets and received TCP packets are never passed to a raw socket. If a process wants to read IP datagrams containing UDP or TCP packets, the packets must be read at the datalink layer, as described in Chapter 29.

UNP1一书中文版的翻译原文是这样的：

接收到UDP分组和TCP分组绝不传递到任何原始套接口。如果一个进程想要读取含有UDP分组或TCP分组的IP数据报，它就必须在数据链路层读取这些分组。

对于中文版的翻译，上文中的“分组”实在是不专业，因为这不是一个准确的术语。读者在看到这个部分后，绝对会很疑惑。分组？何谓分组？是分片的笔误还是组播？笔者自己也是对照了英文原版后才明白中文版的意思。与其用一个模糊的“分组”，还不如直接用“报文”更直截了当。

回到正题，根据UNP1的说法，普通的raw socket是无法收到TCP和UDP的数据包的，除非该套接字是从数据链路层就开始读取数据包的。而实际上Linux内核的实际行为却不是这样的。下面让我们用代码来说明：

```
int raw_local_deliver(struct sk_buff *skb, int protocol)
{
    int hash;
    struct sock *raw_sk;
    /* 根据传输层协议确定

hash桶索引

*/
    hash = protocol & (RAW_HTABLE_SIZE - 1);
    /* 获得该桶的头结点

*/
    raw_sk = sk_head(&raw_v4_hashinfo.ht[hash]);
    /* 当头结点不为空时，才进入

raw_v4_input做进一步检查

*/
    if (raw_sk && !raw_v4_input(skb, ip_hdr(skb), hash)) {
        /* 如果没有找到匹配的原始套接字，则重置

raw_sk为
```

NULL。

```
*/
    raw_sk = NULL;
}
return raw_sk != NULL;
}
```

然后进入raw_v4_input，代码如下：

```
static int raw_v4_input(struct sk_buff *skb, const struct iphdr *iph, int hash)
{
    struct sock *sk;
    struct hlist_head *head;
    int delivered = 0;
    struct net *net;
    /* 与
```

raw_local_deliver不同，因为需要使用到头结点中的内容，所以需要对这个桶上锁，才能保证

在处理这个桶的过程中，所有节点都是有效的。

```
*/
    read_lock(&raw_v4_hashinfo.lock);
    /* 再次检查头结点

*/
    head = &raw_v4_hashinfo.ht[hash];
    if (hlist_empty(head))
        goto out;
    /* 获得网络名称空间

*/
    net = dev_net(skb->dev);
    /* 查询匹配的原始套接字

*/
    sk = __raw_v4_lookup(net, __sk_head(head), iph->protocol,
                        iph->saddr, iph->daddr,
                        skb->dev->ifindex);
    /* 若找到了匹配的原始套接字，则继续处理

*/
    while (sk) {
        delivered = 1;
        /* 如果数据包不是
```

ICMP数据包，或者不是被指定要过滤的

ICMP类型


```

*/
    if (iph->protocol != IPPROTO_ICMP || !icmp_filter(sk, skb)) {
        /* 数据包要发给该套接字，需要

clone一个新的

skb */
        struct sk_buff *clone = skb_clone(skb, GFP_ATOMIC);
        /* 若

clone成功，则调用原始套接字的接收函数

*/
        if (clone)
            raw_rcv(sk, clone);
        /* 继续查询后面的套接字

*/
        sk = __raw_v4_lookup(net, sk_next(sk), iph->protocol,
                             iph->saddr, iph->daddr,
                             skb->dev->ifindex);
    }
out:
    read_unlock(&raw_v4_hashinfo.lock);
    return delivered;
}

```

进入__raw_v4_lookup，代码如下：

```

static struct sock *__raw_v4_lookup(struct net *net, struct sock *sk,
                                   unsigned short num, __be32 raddr, __be32 laddr, int dif)
{
    struct hlist_node *node;
    /* 遍历套接字

*/
    sk_for_each_from(sk, node) {
        struct inet_sock *inet = inet_sk(sk);
        /*
        检查如下几个条件：

        1) 检查名称空间。

        2) 比较协议号。

        3) 如果套接字设置了目的地址且地址相同。

        4) 如果套接字设置了源地址且地址相同。

```

5) 如果套接字绑定了网卡，且网卡相同。

只有当以上五个条件都匹配的时候，该套接字才匹配。

```
*/
if (net_eq(sock_net(sk), net) && inet->inet_num == num &&
    !(inet->inet_daddr && inet->inet_daddr != raddr) &&
    !(inet->inet_rcv_saddr && inet->inet_rcv_saddr != laddr) &&
    !(sk->sk_bound_dev_if && sk->sk_bound_dev_if != dif))
    goto found; /* gotcha */
}
sk = NULL;
found:
return sk;
}
```

在上面的匹配条件中，源地址、目的地址和绑定网卡是原始套接字调用connect、bind等系统调用设置的过滤条件。增加这些过滤条件，一般是为了让应用层减少不必要的消耗，避免过滤不需要过滤的数据包。可以发现，在原始套接字的接收流程中，并没有对TCP和UDP进行任何的限制。也就是说在Linux环境下，普通的原始套接字完全可以接受TCP和UDP的数据包，这与UNP的描述不符。那这是怎么回事呢？因为Stevens大师的UNP针对的是Unix环境的网络编程，而Linux虽然是与Unix兼容的，但在细节的实现上必然与Unix有所不同。需要注意的是，Stevens大师的另一本经典书籍APUE，也是针对Unix环境的介绍，在某些细节上肯定会与Linux环境有一定的出入。

14.6.6 注册传输层协议

在14.5.4节提到的`ip_local_deliver_finish`函数中，内核通过调用`ipprot->handler(skb)`将数据包传递给了正确的传输层协议。对于IPv4协议来说，其传输层协议的处理函数的`handler`是在`inet_init`中添加的。下面是`inet_init`中的部分代码：

```
/* 添加

ICMP协议

*/
if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
    printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
/* 添加

UDP协议

*/
if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
    printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
/* 添加

TCP协议

*/
if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
    printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
#ifdef CONFIG_IP_MULTICAST
/* 添加

IGMP协议

*/
if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
    printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
#endif
```

通过调用`inet_add_protocol`函数，传输层将自己的处理函数添加到了`inet_protos`中，这样就可以在`ip_local_deliver_finish`中调用对应的传输层的处理函数了。

`inet_init`中的另一部分代码如下：

```
for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
    inet_register_protosw(q);
```

这部分代码用于注册`AF_INET`的各种协议，如UDP、TCP等。那为什么`inet`会使用两种不同的方式

来支持传输层协议的注册呢？为何不合并为一个结构呢？在笔者看来，`inet_add_protocol`面向的是底层接口，而`inet_register_protosw`面向的是上层应用，所以将其分为了两个结构。

14.6.7 确定UDP套接字

UDP协议的面向底层接口的处理结构为：

```
static const struct net_protocol udp_protocol = {
    .handler = udp_rcv,
    .err_handler = udp_err,
    .gso_send_check = udp4_ufo_send_check,
    .gso_segment = udp4_ufo_fragment,
    .no_policy = 1,
    .netns_ok = 1,
};
```

因此，如果是UDP数据包，会依次进入udp_rcv→__udp4_lib_rcv，下面来看看__udp4_lib_rcv的相关代码：

```
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable,
                  int proto)
{
    struct sock *sk;
    struct udphdr *uh;
    unsigned short ulen;
    struct rtable *rt = skb_rtable(skb);
    __be32 saddr, daddr;
    struct net *net = dev_net(skb->dev);
    /* 校验数据包至少要有
```

UDP首部大小

```
*/
    if (!pskb_may_pull(skb, sizeof(struct udphdr)))
        goto drop; /* No space for header. */
    /* 得到
```

UDP首部指针

```
*/
    uh = udp_hdr(skb);
    /* 得到
```

UDP数据包长度、源地址、目的地址

```
*/
    ulen = ntohs(uh->len);
    saddr = ip_hdr(skb)->saddr;
    daddr = ip_hdr(skb)->daddr;
    /* 如果
```

UDP数据包长度超过数据包的实际长度，则出错

```
*/
    if (ulen > skb->len)
        goto short_packet;
    /*
    判断协议是否为
```

UDP协议。

也许有的读者会觉得很奇怪，为什么在

UDP的接收函数中还要判断协议是否为

UDP?

因为这个函数还用于处理

UDPLITE协议。

```
*/
if (proto == IPPROTO_UDP) {
    /* 如果是
```

UDP协议，则将数据包的长度更新为

UDP指定的长度，并更新校验和

```
*/
    if (ulen < sizeof(*uh) || pskb_trim_rcsum(skb, ulen))
        goto short_packet;
    /* 因为前面的操作可能会导致
```

skb内存变化，所以需要重新获得

UDP首部指针

```
*/
    uh = udp_hdr(skb);
}
/* 初始化
```

UDP校验和

```
*/
if (udp4_csum_init(skb, uh, proto))
    goto csum_error;
/* 如果路由标志位广播或多播，则表明该
```

UDP数据包为广播或多播

```
*/
if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
    return __udp4_lib_mcast_deliver(net, skb, uh,
        saddr, daddr, udptable);
/* 确定匹配的
```

UDP套接字

```
*/
sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable);
if (sk != NULL){
    /* 找到了匹配的套接字
```

```
*/
    /* 将数据包加入到
```

UDP的接收队列

```
*/
    int ret = udp_queue_rcv_skb(sk, skb);
    sock_put(sk);
    /* a return value > 0 means to resubmit the input, but
     * it wants the return to be -protocol, or 0
     */
    if (ret > 0)
        return -ret;
    return 0;
}
/* 进行
```

xfrm策略检查

```
*/
if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
    goto drop;
/* 重置
```

netfilter信息

```
*/
nf_reset(skb);
/* 检查
```

UDP校验和

```
*/
if (udp_lib_checksum_complete(skb))
    goto csum_error;
/* 若不知道匹配的
```

UDP套接字，则发送

ICMP错误消息

```
*/
UDP_INC_STATS_BH(net, UDP_MIB_NOPTS, proto == IPPROTO_UDPLITE);
icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
/*
 * Hmm. We got an UDP packet to a port to which we
 * don't wanna listen. Ignore it.
 */
kfree_skb(skb);
return 0;
/* 错误处理

*/
.....

}
```

下面来看一下如何匹配UDP套接字，请看__udp4_lib_lookup_skb→__udp4_lib_lookup函数，代码如下：

```
static struct sock *__udp4_lib_lookup(struct net *net, __be32 saddr,
__be16 sport, __be32 daddr, __be16 dport,
int dif, struct udp_table *udptable)
{
    struct sock *sk, *result;
    struct hlist_nulls_node *node;
    unsigned short hnum = ntohs(dport);
    /* 使用目的端口确定
```

hash桶索引

```
*/
unsigned int hash2, slot2, slot = udp_hashfn(net, hnum, udptable->mask);
struct udp_hslot *hslot2, *hslot = &udptable->hash[slot];
int score, badness;
rcu_read_lock();
/* 若该桶的套接字数多于
```

10个，则需要再次定位

```
*/
if (hslot->count > 10) {
    /* 使用目的地址和目的端口确定
```

hash桶索引

```
*/
hash2 = udp4_portaddr_hash(net, daddr, hnum);
slot2 = hash2 & udptable->mask;
/*
UDP套接字表维护了两个
```


hash表:

第一个

hash表，使用端口来索引。

第二个

hash表，使用地址

+端口来索引。

在进行

UDP套接字匹配的时候，优先使用第一个

hash表，因为第一个

hash表使用的是端口进行散

列索引，那么只要端口相同，无论是监听的指定

IP还是任意

IP，都可以在一个桶中进行匹配。但

是由于端口只有

65535种可能，所以可能导致不够分散，一个桶的套接字数会比较多。而第二个

hash表是使用地址

+端口来索引的，因此理论上套接字的分布会比第一个

hash表更加分散。

因此当第一个

hash表对应桶的套接字多于

10个时，内核会尝试去第二个

hash表中进行匹配查找。

```
*/
hslot2 = &udptable->hash2[slot2];
/* 尽管第二个
```

hash表理论上会比第一个

hash表分散，但是如果实际上第二个表的桶中套接字数大于第一个表的桶中套接字数，那么这时还是利用第一个

hash表进行匹配

```
*/
if (hslot->count < hslot2->count)
    goto begin;
/* 在第二个
```

hash表的桶中匹配查找套接字

```
*/
result = udp4_lib_lookup2(net, saddr, sport,
                           daddr, hnum, dif,
                           hslot2, slot2);
if (!result) {
    /* 若利用指定的
```

IP和端口在该桶中没能找到匹配的套接字，则通常使用任意

IP+端口来进行

散列索引

```
*/
hash2 = udp4_portaddr_hash(net, htonl(INADDR_ANY), hnum);
slot2 = hash2 & udptable->mask;
hslot2 = &udptable->hash2[slot2];
/* 还是要与第一个
```

hash桶中的个数进行比较

```
*/
    if (hslot->count < hslot2->count)
        goto begin;
/* 在第二个
```

hash表中使用任意

IP+端口进行匹配查找

```
*/
    result = udp4_lib_lookup2(net, saddr, sport,
                               htonl(INADDR_ANY), hnum, dif,
                               hslot2, slot2);
    rcu_read_unlock();
    return result;
}
begin:
    result = NULL;
    badness = -1;
/* 在第一个
```

hash表的桶中进行查找

```
*/
sk_nulls_for_each_rcu(sk, node, &hslot->head) {
/* 计算该套接字的匹配得分

*/
    score = compute_score(sk, net, saddr, hnum, sport,
                           daddr, dport, dif);
/* 保证匹配得分最高的套接字为最终结果

*/
    if (score > badness) {
        result = sk;
        badness = score;
    }
}
/*
检查在查找的过程中，是否遇到了某个套接字被移到另外一个桶内的情况。
```

这时，需要重新进行匹配。

```
*/
if (get_nulls_value(node) != slot)
    goto begin;
/* 找到了匹配的套接字
```

```
*/
if (result) {
/* 增加套接字引用计数
```

```

*/
    if (unlikely(!atomic_inc_not_zero_hint(&result->sk_refcnt, 2)))
        result = NULL;
    /* 再次计算套接字得分，如小于最大分数，则重新匹配查找。之所以做二次检查，也是为了防止在

        匹配与增加引用的过程中，套接字发生变化。

    */
    else if (unlikely(compute_score(result, net, saddr, hnum, sport,
        daddr, dport, dif) < badness)) {
        sock_put(result);
        goto begin;
    }
}
rcu_read_unlock();
return result;
}

```

从上面的代码中可以看到，匹配UDP套接字的关键在于对应套接字的匹配得分。第一个hash表的得分计算函数为compute_score。

```

static inline int compute_score(struct sock *sk, struct net *net, __be32 saddr,
    unsigned short hnum,
    __be16 sport, __be32 daddr, __be16 dport, int dif)
{
    int score = -1;
    /* 比较名称空间，端口等

```

```

*/
    if (net_eq(sock_net(sk), net) && udp_sk(sk)->udp_port_hash == hnum &&
        !ipv6_only_sock(sk)) {
        struct inet_sock *inet = inet_sk(sk);
        /* 若套接字指明为

```

PF_INET, 则加

1分

```

*/
    score = (sk->sk_family == PF_INET ? 1 : 0);
    /* 套接字绑定了接收地址

```

```

*/
    if (inet->inet_rcv_saddr) {
        /* 如果数据包的目的地址与绑定接收地址不符，则分数为

```

-1, 相同则增加

2分。

```

*/
    if (inet->inet_rcv_saddr != daddr)

```

```

        return -1;
        score += 2;
    }
    /* 套接字设置了对端目的地址

```

```

*/
    if (inet->inet_daddr) {
        /* 如果数据包的源地址与设置的目的地址不同，则分数为

```

-1，相同则增加

2分

```

*/
        if (inet->inet_daddr != saddr)
            return -1;
        score += 2;
    }
    /* 套接字设置了对端目的端口

```

```

*/
    if (inet->inet_dport) {
        /* 如果数据包的源端口与设置的目的端口不同，则分数为

```

-1，相同则增加

2分

```

*/
        if (inet->inet_dport != sport)
            return -1;
        score += 2;
    }
    /* 套接字绑定了网卡

```

```

*/
    if (sk->sk_bound_dev_if) {
        /* 如果接受数据包的网卡与绑定网卡不同，则分数为

```

-1，相同则增加

2分

```

*/
        if (sk->sk_bound_dev_if != dif)
            return -1;
        score += 2;
    }
}
return score;
}

```

对于第二个hash，其匹配分数计算函数为compute_score2，算法与compute_score基本相同。总的来

说UDP的套接字匹配有以下几个条件：

- 接收端口：必须匹配。
- 接收地址：如绑定了则必须匹配，分值为2分。
- 对端目的地址：如设置了则必须匹配，分值为2分。
- 对端目的端口：如设置了则必须匹配，分值为2分。
- 网卡：如绑定了则必须匹配，分值为2分。
- 套接字设置了PF_INET协议族，分值为1分。

根据上面的规则，匹配分值最高的套接字就为选中的UDP套接字，然后内核会将这个数据包加入到该UDP套接字的接收队列中。也就是说，即使数据包可以匹配多个UDP套接字（这是很有可能的），但是最终也只有一个最匹配的套接字会被选中，并且只有这个套接字可以收到数据包。

有一些开发人员想使用套接字的SO_REUSEADDR选项，让多个套接字绑定同一个地址或端口，然后让独立的线程或进程负责一个套接字的处理，希望利用这样的设计来提高服务的响应速度。这里面有个想当然的认为，当多个套接字负责同一个地址和端口的数据包接收时，它们可以分担负载。然而从上面的源码分析中，我们可以发现这样的设计方案是达不到预期效果的。因为内核在进行套接字的匹配时，对于绑定相同地址和端口的多个套接字，每次只会命中同一个套接字。结果在上面的设计中，只有一个套接字会收到数据包，也就说最后只有一个线程或进程在处理数据包。

不过Linux内核在3.9版本中引入了一个新的套接字选项SO_REUSEPORT用于解决上面的问题。当多个套接字绑定于同一个地址和端口时，并启用了SO_REUSEPORT时，内核会自动在这几个套接字之间做负载均衡，保证对应的数据包能尽量平均地分配到不同的套接字上。

14.6.8 确定TCP套接字

TCP面向底层接口的处理结构为：

```
static const struct net_protocol tcp_protocol = {
    .handler = tcp_v4_rcv,
    .err_handler = tcp_v4_err,
    .gso_send_check = tcp_v4_gso_send_check,
    .gso_segment = tcp_tso_segment,
    .gro_receive = tcp4_gro_receive,
    .gro_complete = tcp4_gro_complete,
    .no_policy = 1,
    .netns_ok = 1,
};
```

那么，如果是TCP数据包，则会进入tcp_v4_rcv，代码如下：

```
int tcp_v4_rcv(struct sk_buff *skb)
{
    const struct iphdr *iph;
    const struct tcphdr *th;
    struct sock *sk;
    int ret;
    struct net *net = dev_net(skb->dev);
    /* 丢弃不是发给自己的数据包

    */
    if (skb->pkt_type != PACKET_HOST)
        goto discard_it;
    /* Count it even if it's bad */
    TCP_INC_STATS_BH(net, TCP_MIB_INSEGS);
    /* 检查数据包至少要有
```

TCP固定首部的大小

```
*/
    if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
        goto discard_it;
    /* 获得
```

TCP首部地址

```
*/
    th = tcp_hdr(skb);
    /* 检查
```

TCP的数据偏移量是否合法

，不能小于

TCP固定首部的大小

```
*/
    if (th->doff < sizeof(struct tcphdr) / 4)
```

```

        goto bad_packet;
/* 检查数据包的大小是否满足

```

TCP指定的数据偏移位置

```

*/
if (!pskb_may_pull(skb, th->doff * 4))
    goto discard_it;
/* 如果需要检查校验和，则进行校验和初始化

*/
if (!skb_csum_unnecessary(skb) && tcp_v4_checksum_init(skb))
    goto bad_packet;
/* 因为

```

skb可能会重新申请，所以需要重新得到

TCP首部

```

*/
th = tcp_hdr(skb);
iph = ip_hdr(skb);
/* 根据

```

TCP报文信息，设置

skb的

TCP控制块

```

*/
TCP_SKB_CB(skb)->seq = ntohl(th->seq);
TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin + skb->len - th->doff * 4);
TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
TCP_SKB_CB(skb)->when = 0;
TCP_SKB_CB(skb)->ip_dsfield = ipv4_get_dsfield(iph);
TCP_SKB_CB(skb)->sacked = 0;
/* 查找匹配的

```

TCP套接字

```

*/
sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
if (!sk)
    goto no_tcp_socket;
/* 找到匹配的套接字并开始处理，因为不是本文重点，因此省略后面的分析

```

```

*/
process:
.....

```



```
}
```

进入__inet_lookup_skb->__inet_lookup。

```
static inline struct sock *__inet_lookup(struct net *net,
                                         struct inet_hashinfo *hashinfo,
                                         const __be32 saddr, const __be16 sport,
                                         const __be32 daddr, const __be16 dport,
                                         const int dif)
{
    u16 hnum = ntohs(dport);
    /* 先在已连接的套接字中进行查找

    */
    struct sock *sk = __inet_lookup_established(net, hashinfo,
                                                saddr, sport, daddr, hnum, dif);
    /* 优先使用已连接的套接字, 若没有找到, 则在监听的套接字中进行查找

    */
    return sk ? : __inet_lookup_listener(net, hashinfo, daddr, hnum, dif);
}
```

对于TCP来说, 套接字的匹配分为两部分: 一个是匹配已经建立连接的, 另外一个匹配是匹配监听状态的套接字。我们先来看看前者, 即已经建立连接的, 进入__inet_lookup_established, 代码如下:

```
struct sock *__inet_lookup_established(struct net *net,
                                       struct inet_hashinfo *hashinfo,
                                       const __be32 saddr, const __be16 sport,
                                       const __be32 daddr, const u16 hnum,
                                       const int dif)
{
    INET_ADDR_COOKIE(acookie, saddr, daddr)
    const __portpair ports = INET_COMBINED_PORTS(sport, hnum);
    struct sock *sk;
    const struct hlist_nulls_node *node;
    /* Optimize here for direct hit, only listening connections can
     * have wildcards anyways.
     */
    /* 根据目的地址、目的端口、源地址、源端口计算得到已经连接了
```

hash表的桶索引

```
    */
    unsigned int hash = inet_ehashfn(net, daddr, hnum, saddr, sport);
    unsigned int slot = hash & hashinfo->ehash_mask;
    struct inet_ehash_bucket *head = &hashinfo->ehash[slot];
    rcu_read_lock();
begin:
    /* 遍历该桶节点
```

```
    */
    sk_nulls_for_each_rcu(sk, node, &head->chain) {
        /*
         * 比较源地址、目的地址、源端口、目的端口及接收网卡。
```

细心的读者会发现这里有两个参数

acookie和

ports, 这两个参数用于加速匹配。在

64位机器上,

acookie为源地址和目的地址合成的

64位整数, 在

32位机器上

acookie并无意义。

ports为源端

口和目的端口合成的

32位整数。通过直接比较组合的整数, 可以加速匹配。

```
*/
if (INET_MATCH(sk, net, hash, acookie,
               _saddr, daddr, ports, dif)) {
    /* 增加套接字引用计数

*/
    if (unlikely(!atomic_inc_not_zero(&sk->sk_refcnt)))
        goto begintw;
    /*
       再次检测, 防止套接字在

INET_MATCH和增加计数之间被改变

*/
    if (unlikely(!INET_MATCH(sk, net, hash, acookie,
                             saddr, daddr, ports, dif))) {
        sock_put(sk);
        goto begin;
    }
    goto out;
}
/*
 * if the nulls value we got at the end of this lookup is
 * not the expected one, we must restart lookup.
 * We probably met an item that was moved to another chain.
 */
if (get_nulls_value(node) != slot)
    goto begin;
begintw:
/* 如果在已经连接的
```

hash表中找不到对应的套接字，则需要到连接为

TIME_WAIT状态的

hash表中查找

套接字，原理与上面相同。这说明

TIME_WAIT状态的连接如果依然存在，则会优先于监听套接字。

```
*/
sk_nulls_for_each_rcu(sk, node, &head->twchain) {
    if (INET_TW_MATCH(sk, net, hash, acookie,
                      saddr, daddr, ports, dif)) {
        if (unlikely(!atomic_inc_not_zero(&sk->sk_refcnt))) {
            sk = NULL;
            goto out;
        }
        /* 与上面一样，需要再次进行匹配检查 */
    }

    if (unlikely(!INET_TW_MATCH(sk, net, hash, acookie,
                                saddr, daddr, ports, dif))) {
        sock_put(sk);
        goto begintw;
    }
    goto out;
}
/* 省略 */

*/
.....

}
```

如果在已经连接和TIME_WAIT状态的hash表中，都没有找到匹配的套接字。这时就需要到监听hash表中查找匹配的套接字，代码如下：

```
struct sock *__inet_lookup_listener(struct net *net,
                                   struct inet_hashinfo *hashinfo,
                                   const __be32 daddr, const unsigned short hnum,
                                   const int dif)
{
    struct sock *sk, *result;
    struct hlist_nulls_node *node;
    /* 根据源端口计算监听
```

hash表对应的桶索引

```

*/
unsigned int hash = inet_lhashfn(net, hnum);
struct inet_listen_hashbucket *ilb = &hashinfo->listening_hash[hash];
int score, hiscore;
rcu_read_lock();
begin:
result = NULL;
hiscore = -1;
/* 遍历该桶中的套接字, 与

```

UDP相似, 得分最高的套接字为匹配套接字

```

*/
sk_nulls_for_each_rcu(sk, node, &ilb->head) {
    /* 计算套接字的得分

*/
    score = compute_score(sk, net, hnum, daddr, dif);
    if (score > hiscore) {
        result = sk;
        hiscore = score;
    }
}
/*
 * if the nulls value we got at the end of this lookup is
 * not the expected one, we must restart lookup.
 * We probably met an item that was moved to another chain.
 */
if (get_nulls_value(node) != hash + LISTENING_NULLS_BASE)
    goto begin;
/* 找到了匹配的套接字

*/
if (result) {
    /* 增加套接字引用计数

*/
    if (unlikely(!atomic_inc_not_zero(&result->sk_refcnt)))
        result = NULL;
    /* 需要再次检查该套接字的得分

*/
    else if (unlikely(compute_score(result, net, hnum, daddr,
        dif) < hiscore)) {
        sock_put(result);
        goto begin;
    }
}
rcu_read_unlock();
return result;
}

```

匹配TCP监听套接字的流程与UDP基本相同, 都是在计算套接字的得分。下面我们来看一下TCP监听套接字的得分计算函数compute_score。

```

static inline int compute_score(struct sock *sk, struct net *net,
    const unsigned short hnum, const __be32 daddr,
    const int dif)
{
    int score = -1;
    struct inet_sock *inet = inet_sk(sk);
    /* 必须匹配名称空间和目的端口

*/
    if (net_eq(sock_net(sk), net) && inet->inet_num == hnum &&

```

```

        !ipv6_only_sock(sk)) {
            be32 rcv_saddr = inet->inet_rcv_saddr;
        /* 若协议族为

```

PF_INET, 则得分加

```

1 */
    score = sk->sk_family == PF_INET ? 1 : 0;
    /*
    若套接字指定了接收地址, 则接收地址必须与目的地址相同。

```

不同则不匹配, 相同则得分加

2。

```

    */
    if (rcv_saddr) {
        if (rcv_saddr != daddr)
            return -1;
        score += 2;
    }
    /*
    如果套接字绑定了网卡, 则接收网卡必须相同。

```

不同则不匹配, 相同则得分加

2。

```

    */
    if (sk->sk_bound_dev_if) {
        if (sk->sk_bound_dev_if != dif)
            return -1;
        score += 2;
    }
}
return score;
}

```

这个函数的名字和功能均与UDP的相同, 但是其实现代码却略有不同。大家可以发现, TCP的compute_score不会对源端信息进行任何检查, 如源地址、源端口等。为什么会这样呢? 原因在于, TCP的compute_score是用于监听套接字的匹配的。这就意味着这是TCP连接的第一个数据包即SYN包, 属于连接初始化阶段, 这时自然无须对源端信息进行任何的检查和匹配。在本机回复了SYN+ACK后, 本机会创建已连接套接字并插入到已连接hash表中。这样在此连接后面的数据包, 就会在已连接的hash表中找到匹配的套接字, 而不会再进入监听套接字的匹配。

第15章 编写安全无错代码

通过前面的章节，主要是学习和分析在Linux环境下不同方面的系统调用及内核实现。本章则将分享笔者多年编程的一些经验，主要是从基础概念出发，介绍一些编码细节，这些细节看上去有些分散，有点奇技淫巧的味道，但笔者的主要目的是为了让大家从心里明白编写安全无错代码的不易。要对代码有敬畏之心，才能真正驾驭代码，写出健壮的程序。

15.1 不要用memcmp比较结构体

比较两个结构体时，若结构体中含有大量的成员变量，为了方便，程序员往往会直接使用memcmp对这两个结构体进行比较，以避免对每个成员进行分别比较。这样的代码写起来比较简单，然而却很可能深藏隐患。请看下面的示例代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct padding_type {
    short m1;
    int m2;
} padding_type_t;
int main()
{
    padding_type_t a = {
        .m1 = 0,
        .m2 = 0,
    };
    padding_type_t b;
    memset(&b, 0, sizeof(b));
    if (0 == memcmp(&a, &b, sizeof(a))) {
        printf("Equal!\n");
    }
    else {
        printf("No equal!\n");
    }
    return 0;
}
```

大家先想一下，结果会是什么？一起来看看最终的结果：

```
[fgao@ubuntu chapter15]#gcc -Wall 15_1_cmp_struct.c
[fgao@ubuntu chapter15]#./a.out
No equal!
```

为什么会是这样的结果呢？有经验的读者立刻就会反应过来：这是由于对齐造成的。

没错！就是因为struct padding_type->m1的类型是short类型，而m2的类型是int类型。根据自然对齐规则，struct padding_type需要进行4字节对齐。因此编译器会在m1后面插入两个padding字节，而这两个字节的内容却是“随机”的。结构体b由于调用了memset对整个结构体占用的内存进行了清零，其padding的值自然就为0。这样，当使用memcmp对两个结构体进行比较时，结论就是不相同了，即返回值不为0。

所以，除非在项目中可以保证所有的结构体都会使用memset来进行初始化（这个是很难保证的），否则就不要直接使用memcmp来比较结构体。

15.2 有符号数和无符号数的移位区别

在代码规范中一般都会要求，如果没有符号要求，则尽量使用无符号整数，避免使用有符号整数。因为有符号整数在一些常见的操作中，将表现出与无符号整数大相径庭的行为。本节将展示有符号整数与无符号整数在移位操作上的区别。来看一个示例：

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int a = 0x80000000;
    unsigned int b = 0x80000000;
    printf("a right shift value is 0x%X\n", a >> 1);
    printf("b right shift value is 0x%X\n", b >> 1);
    return 0;
}
```

其输出结果为：

```
[fgao@ubuntu chapter15]#gcc -Wall 15_2_sign_shift.c
[fgao@ubuntu chapter15]#./a.out
a right shift value is 0xC0000000
b right shift value is 0x40000000
```

为了了解为什么会产生这样的结果，请看其汇编代码：

```
Dump of assembler code for function main:
0x0804841d <+0>:    push    %ebp
0x0804841e <+1>:    mov     %esp,%ebp
0x08048420 <+3>:    and     $0xffffffff,%esp
0x08048423 <+6>:    sub     $0x20,%esp
0x08048426 <+9>:    movl    $0x80000000,0x18(%esp)
0x0804842e <+17>:   movl    $0x80000000,0x1c(%esp)
0x08048436 <+25>:   mov     0x18(%esp),%eax
0x0804843a <+29>:   sar     %eax
0x0804843c <+31>:   mov     %eax,0x4(%esp)
0x08048440 <+35>:   movl    $0x8048500, (%esp)
0x08048447 <+42>:   call    0x80482f0 <printf@plt>
0x0804844c <+47>:   mov     0x1c(%esp),%eax
0x08048450 <+51>:   shr     %eax
0x08048452 <+53>:   mov     %eax,0x4(%esp)
0x08048456 <+57>:   movl    $0x804851d, (%esp)
0x0804845d <+64>:   call    0x80482f0 <printf@plt>
0x08048462 <+69>:   mov     $0x0,%eax
0x08048467 <+74>:   leave
0x08048468 <+75>:   ret
End of assembler dump.
```

0x80000000在内存或寄存器中的布局如图15-1所示。



图15-1 0x80000000在内存或寄存器中的布局

其中第一位“1”即符号位。

0x0804843a地址对应的是a>>1的汇编代码，sar为算术右移，其使用符号位补位。对于此例，即用1补位。0x08048450地址对应的是b>>1的汇编代码，shr为逻辑右移，其使用0补位。这就造成了最终结果

的不同。

15.3 数组和指针

对于这个标题，可能很多读者都会认为数组和指针，几乎没有什么区别。确实，在大多数的情况下，数组和指针的区别并不大，甚至可以互换。然而，这两者实际上是有本质区别的。而这个区别也会导致并不是所有的情况下，两者都可以互换。同样来看一个示例：

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int array[4] = {0};
    int *pointer = NULL;
    int value = 0;
    value = array;
    value = &array;
    value = array[0];
    value = &array[0];
    value = point;
    value = &point;
    return 0;
}
```

对其反汇编，来分析数组和指针的本质。分析的过程将直接写在反汇编的代码中：

Dump of assembler code for function main:
0x080483ed <+0>: push %ebp
0x080483ee <+1>: mov %esp,%ebp
0x080483f0 <+3>: sub \$0x20,%esp
/*
下面

4行对应

C代码

int array[4] = {0};
这里说明数组只是一个同类型变量的内存空间的集合。

这个例子中，

array是在栈上申请了

4个整型变量的空间。

*/
0x080483f3 <+6>: movl \$0x0,-0x10(%ebp)
0x080483fa <+13>: movl \$0x0,-0xc(%ebp)
0x08048401 <+20>: movl \$0x0,-0x8(%ebp)
0x08048408 <+27>: movl \$0x0,-0x4(%ebp)
/*
这行对应的

C代码为

```
int *pointer = NULL.  
    这说明指针本身也是一个变量，同样占用了栈空间。
```

32位机器上，其占用

4字节。

数组和指针对比，其占用的空间实际上是数组中元素占用的空间之和。

本例中，即

array[0],array[1],array[2],array[3], 而

array本身实际上更像是一个

label。

```
*/  
0x0804840f <+34>:    movl    $0x0,-0x18(%ebp)  
/* 这行对应的
```

C代码为

```
int value = 0; */  
0x08048416 <+41>:    movl    $0x0,-0x14(%ebp)  
/*  
    下面两行对应的
```

C代码是

```
value = array;  
    这两行汇编代码是指取得
```

array首元素的地址并将其赋给

eax寄存器，然后再将

eax的值赋给

value。

lea是汇编中的取址操作。

```
*/
0x0804841d <+48>:    lea    -0x10(%ebp),%eax
0x08048420 <+51>:    mov    %eax,-0x14(%ebp)
/*
下面两行代码对应的
```

C代码为

value = &array。

其仍然是取

array首元素的地址赋值给

value。

```
*/
0x08048423 <+54>:    lea    -0x10(%ebp),%eax
0x08048426 <+57>:    mov    %eax,-0x14(%ebp)
/*
这两行代码对应
```

value=array[0]。

注意这里使用的是

mov汇编指令，即将值赋给

eax。

```
*/
0x08048429 <+60>:    mov    -0x10(%ebp),%eax
0x0804842c <+63>:    mov    %eax,-0x14(%ebp)
/*
对应的代码为
```

value = &array[0]。

从汇编指令中可以明确看出，

array、

&array、

&array[0]，实际上都是同一个地址。

```
*/
0x0804842f <+66>:    lea    -0x10(%ebp),%eax
0x08048432 <+69>:    mov    %eax,-0x14(%ebp)
```

/*
对应的代码是

value = pointer。

注意这里使用的是

mov指令而不是

lea指令。是将指针

int *pointer的值

0赋值给

value。

```
*/
0x08048435 <+72>:    mov    -0x18(%ebp),%eax
0x08048438 <+75>:    mov    %eax,-0x14(%ebp)
```

/*
对应的代码是

value = &pointer;

是将

int *pointer的地址赋值给

value.

```
*/
0x0804843b <+78>:    lea    -0x18(%ebp),%eax
0x0804843e <+81>:    mov    %eax,-0x14(%ebp)
0x08048441 <+84>:    mov    $0x0,%eax
0x08048446 <+89>:    leave
0x08048447 <+90>:    ret
End of assembler dump.
```

通过上面的汇编代码，我们可以深入地理解C语言中的指针和数组的真正含义。要认识到指针其实就是一个变量，只不过这个变量是用于保存地址的（实际上也可以保存其他内容，如一个整数），或者说它保存的值可以被视为地址。因为指针类型可以合法地使用“*”运算符，做提领运算。而这个提领运算，其实就是将变量的值视为一个地址，然后从这个地址中读取值。

为了加深对指针本质的理解，请看下面的例子：

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    short *p1 = 0;
    int **p2 = 0;
    ++p1;
    ++p2;
    printf("p1 = %d, p2 = %d\n", p1, p2);
    return 0;
}
```

这是我很喜欢的一道题目。大家可以想一下，这个程序是否会崩溃？如果崩溃，原因是什么？如果不崩溃，其输出结果是什么？

如果真正理解了指针，看完代码，就可以迅速地说出最终的结果。如果你还在犹豫，那就说明你对指针的理解还不够透彻。

其输出结果为：

```
[fgao@ubuntu chapter14]# ./a.out
p1 = 2, p2 = 4
```

15.4 再论数组首地址

15.3节中，通过汇编代码，我们知道array、&array和&array[0]的地址是相同的，那么它们三者是否有相同的含义呢？请看下面的示例代码：

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int a[2][3];
    printf("&a[0][0] address is 0x%X\n", &a[0][0]);
    printf("&a[0][0]+1 address is 0x%X\n", &a[0][0]+1);
    printf("size of pointer step is 0x%X\n", sizeof(*(&a[0][0])));
    printf("\n");
    printf("&a[0] address is 0x%X\n", &a[0]);
    printf("&a[0]+1 address is 0x%X\n", &a[0]+1);
    printf("size of pointer step is 0x%X\n", sizeof(*(&a[0])));
    printf("\n");
    printf("a address is 0x%X\n", a);
    printf("a+1 address is 0x%X\n", a+1);
    printf("size of pointer step is 0x%X\n", sizeof(*a));
    printf("\n");
    printf("&a address is 0x%X\n", &a);
    printf("&a+1 address is 0x%X\n", &a+1);
    printf("size of pointer step is 0x%X\n", sizeof(*(&a)));
    printf("\n");
    return 0;
}
```

大家可以先想一下其运行结果是什么，然后再看下面的结果：

```
[fgao@ubuntu chapter15]# ./a.out
&a[0][0] address is 0xBF903D48
&a[0][0]+1 address is 0xBF903D4C
size of pointer step is 0x4
&a[0] address is 0xBF903D48
&a[0]+1 address is 0xBF903D54
size of pointer step is 0xC
a address is 0xBF903D48
a+1 address is 0xBF903D54
size of pointer step is 0xC
&a address is 0xBF903D48
&a+1 address is 0xBF903D60
size of pointer step is 0x18
```

从输出上看，可以发现&a[0][0]、&a[0]、a，还有&a的地址值都是相同的，然而其步进1即地址+1的值却完全不同。

为什么会是这样呢？因为尽管这几个变量的地址相同，但是其变量类型却是不同的：

- &a[0][0]的类型是int*pointer，所以步长为4字节。
- &a[0]的类型为int（*pointer）[3]，所以步长为12字节。
- a的类型也为int（*pointer）[3]，所以其步长也为12字节。
- &a的类型为int（*pointer）[2][3]，所以其步长为24字节。

15.5 “神奇”的整数类型转换

整数类型转换经常被当作笔试题目之一，大家可能会觉得那样的题目很简单，也许同样会觉得本节也没什么难度。请大家先耐心看一下下面的示例：

```
#include <stdlib.h>
#include <stdio.h>
#define PRINT_COMPARE_RESULT(a, b) \
    if (a > b) { \
        printf("#a > #b\n"); \
    } else if (a < b) { \
        printf("#a < #b\n"); \
    } else { \
        printf("#a = #b\n"); \
    }
int main(void)
{
    signed int a = -1;
    unsigned int b = 2;
    signed short c = -1;
    unsigned short d = 2;
    PRINT_COMPARE_RESULT(a, b);
    PRINT_COMPARE_RESULT(c, d);
    return 0;
}
```

大多数同学可能都遇到过这类将a和b进行比较的题目，结果是a>b，原因也很简单明确：当signed int和unsigned int进行比较时，signed int会被转换为unsigned int。-1的值即0xFFFFFFFF，就被视为无符号整数的最大值，因此a>b。然而对于c和d来说，其类型分别是signed short和unsigned short，那么结果又会是什么呢？请看下面的输出：

```
[fgao@ubuntu chapter15]# ./a.out
a > b
c < d
```

是不是感觉有些意外？为什么仅仅从int变为short，其结果就截然不同了呢？

原因在于C标准规定，当进行整数提升时，如果int类型可以表示原始类型的所有值时，它就被转换为int类型；不然则被转换为unsigned int。所以当c和d进行比较时，c和d的类型分别是short和unsigned short，那么它们就会被转换为int类型，则实际是对(int)-1和(int)2进行比较，结果自然是c<d。

15.6 小心volatile的原子性误解

关于volatile的说明，是一个老生常谈的问题。其定义很简单，可以理解为易变的，防止编译器对其优化。因此其用途一般有以下三种：

- 外部设备寄存器映射后的内存——因为外部寄存器随时可能由于外部设备的状态变化而改变，因此映射后的内存需要用volatile来修饰。
- 多线程或异步访问的全局变量。
- 嵌入式编程——防止编译器对其优化。

对第1种和第3种的用途大家基本上都不会有什么误解，但经常会错误地理解第2种情况：认为int类型的加减操作是原子的，因此在使用了volatile后，就无须使用锁来进行竞争保护了。比如下面这样的代码：

```
static volatile int counter = 0;
void add_counter(void)
{
    ++counter;
}
```

其反汇编代码为：

```
add_counter:
pushl %ebp
movl %esp, %ebp
movl counter, %eax
addl $1, %eax
movl %eax, counter
popl %ebp
ret
```

上面的汇编代码，首先是将counter的值保存到eax寄存器，然后对eax进行加1操作，最后再将eax的值保存到counter中。这样，++counter就绝不可能是原子操作了，必须使用锁保护。

那么volatile对于变量来说，究竟有什么样的效果呢？下面的代码对上面的代码进行了一些修改：

```
static int counter = 0;
void add_counter(void)
{
    for (; counter != 10;) {
        ++counter;
    }
}
```

用gcc-S-O 15_6_volatile.c生成对应的汇编代码：

```
add_counter:
.LFB0:
.cfi_startproc
```

```
        movl    counter, %eax
        cmpl   $10, %eax
        je     .L1
.L4:    addl    $1, %eax
        cmpl   $10, %eax
        jne    .L4
        movl    $10, counter
.L1:    rep ret
        .cfi_endproc
.LFE0:
```

从上面的汇编代码可以清晰地看出，在进入`add_counter`后，首先会将`counter`的值赋给`eax`寄存器，然后`eax`进行加1操作，再与立即数10进行比较。也就是说，`for`循环的C代码只涉及`eax`寄存器，而不会对`counter`进行任何访问。

接下来，对`counter`添加上`volatile`修饰符：

```
static volatile int counter = 0;
void add_counter(void)
{
    for (; counter != 10; ) {
        ++counter;
    }
}
```

然后生成汇编代码`gcc-S-O 15_6_volatile2.c`:

```
add_counter:
.LFB0:
        .cfi_startproc
        movl    counter, %eax
        cmpl   $10, %eax
        je     .L1
.L3:    movl    counter, %eax
        addl    $1, %eax
        movl    %eax, counter
        movl    counter, %eax
        cmpl   $10, %eax
        jne    .L3
.L1:    rep ret
        .cfi_endproc
```

与没有`volatile`的汇编代码相比，其差异很明显。使用了`volatile`之后，与`counter`的自增操作对应的汇编代码，每次都要重新从`counter`读取值，再将其赋值给`eax`寄存器。

现在对`volatile`的理解就比较深刻了。`volatile`只能保证在访问该变量时，每次都是从内存中读取最新值，并不会使用寄存器中缓存的值。而对该变量的修改，`volatile`并不提供原子性的保证。

15.7 有趣的问题：“x==x”何时为假？

看到这个题目，大家可能会想到一些比较另类的方法，比如使用宏定义，或者用高级语言中的操作符重载之类的。但如果说要使用最原始的C语言表达式，那么什么时候“x==x”会是假呢？请看下面的代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    float x = 0xffffffff;
    if (x == x) {
        printf("Equal\n");
    }
    else {
        printf("Not equal\n");
    }
    if (x >= 0) {
        printf("x(%f) >= 0\n", x);
    }
    else if (x < 0) {
        printf("x(%f) < 0\n", x);
    }
    int a = 0xffffffff;
    memcpy(&x, &a, sizeof(x));
    if (x == x) {
        printf("Equal\n");
    }
    else {
        printf("Not equal\n");
    }
    if (x >= 0) {
        printf("x(%f) >= 0\n", x);
    }
    else if (x < 0) {
        printf("x(%f) < 0\n", x);
    }
    else {
        printf("Surprise x(%f)!!!\n", x);
    }
    return 0;
}
```

gcc-Wall 15_7_float.c编译并执行。输出结果如下所示：

```
[fgao@ubuntu chapter15]# ./a.out
Equal
x(4294967296.000000) >= 0
Not equal
Surprise x(-nan)!!!
```

这样的结果是不是有些意外呢？

简单解释一下其中的原因：

·当float x=0xffffffff时，将整数赋值给一个浮点数，由于float和int都占用了4字节，但浮点数的存储格式与整数不同，其需要一定的数位来作为小数位，所以float的表示范围要小于int。这里涉及了C语言中的类型转换。

·当整数转换为浮点数时，尽管数值会有所变化，但结果一定是一个合法的浮点值。所以x一定等于x，且x不是大于等于0，就是小于0。

- 当使用`memcpy`将`0xff`填充到`x`的地址时，这时保证了`x`储存的一定是`0xffffffff`，但很可惜它不是一个合法的浮点值，而是一个特殊值NaN。

- 作为一个非法的浮点数NaN，当它与任何数值相比较时，都会返回假。所以就有了比较意外的结果`x==x`为假，`x`即不大于0，不小于0，也不等于0。

15.8 小心浮点陷阱

15.7节通过“ $x==x$ ”为假这个问题，引出了一个特殊的浮点值NaN。本节将挖掘出更多的浮点陷阱。

15.8.1 浮点数的精度限制

浮点数的存储格式与整数完全不同。大部分的实现采用的是IEEE 754标准，float类型是1个sign bit、8个exponent bits和23个mantissa bits。而double类型是1个sign bit、11个exponent bits和52个mantissa bits。关于浮点数是如何表示小数部分的，大家可以自行参考维基百科。简单来说，小数部分是依靠2的负多少次方来近似表示的，因此浮点数存在精度的问题，对浮点数进行比较时，要使用范围比较。

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    float x = 0.123-0.11-0.013;
    if (x == 0) {
        printf("x is 0!\n");
    }
    if (-0.0000000001 < x && x < 0.0000000001) {
        printf("x is in 0 range!\n");
    }
    return 0;
}
```

编译输出：

```
[fgao@ubuntu chapter15]#gcc -Wall 15_8_float1.c
[fgao@ubuntu chapter15]#./a.out
x is in 0 range!
```

从数学的角度看，float x=0.123-0.11-0.013，得到的一定是0。但对于浮点数来说，因为其不能精确地表示小数，因此x最终的结果是一个趋近于0的值。故而不能用0和x直接进行比较，而是要使用一个范围来确定x是否为0。

15.8.2 两个特殊的浮点值

浮点数有两个特殊的值，除了前面的NaN（Not a Number），还有一个infinite即无限。15.7节中，使用memcpy构造了一个NaN的浮点数。可能有人会问，平常有谁会用memcpy去填充浮点数呢？因此我不可能遇到NaN。那么，请看下面的示例：

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    float x = 1/0.0;
    printf("x is %f\n", x);
    x = 0/0.0;
    printf("x is %f\n", x);
    return 0;
}
```

其输出结果为：

```
[fgao@ubuntu chapter15]#./a.out
x is inf
x is -nan
```

当1除以0.0时，得到的是infinite，而用0除以0.0时，得到的就是NaN。虽然这里完全只是一则普通的除法运算，但也会产生NaN的情况。

那么当使用除法运算时，对除数进行检查，保证其不为0.0，是否就可以避免NaN了？再看下面的代码：

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    float x;
    while (1) {
        scanf("%f", &x);
        printf("x is %f\n", x);
    }
    return 0;
}
```

编译执行：

```
[fgao@ubuntu chapter15]#gcc -Wall 15_8_float3.c
[fgao@ubuntu chapter15]#./a.out
inf
x is inf
nan
x is nan
```

上面的代码中使用了scanf来得到用户输入的浮点数。令人惊讶的是，scanf不仅接受inf和nan的输入，并将其视为浮点数的两种特殊值。那么对于UI程序来说，当遇到浮点数值的时候，我们必须首先判断其是否为合法的浮点值。笔者就遇到过一个开源库返回的浮点数为NaN的情况。令人高兴的是，C

库提供了两个库函数`isinf`和`isnan`，分别用于判断浮点数是否为`infinite`和`NaN`。

15.9 Intel移位指令陷阱

假设操作平台为32位平台，请看下面的示例代码：

```
#include <stdio.h>
int main()
{
#define MOVE_CONSTANT_BITS 32
    unsigned int move_step=MOVE_CONSTANT_BITS;
    unsigned int value1 = 1ul << MOVE_CONSTANT_BITS;
    printf("value1 is 0x%X\n", value1);
    unsigned int value2 = 1ul << move_step;
    printf("value2 is 0x%X\n", value2);
    return 0;
}
```

上面的代码中，value1使用立即数32进行左移，而value2使用一个变量move_step进行左移，且move_step的值也是32。那么问题来了，最后value1和value2的值是什么？我相信大部分人都会说最后两个值是一样的，都是0。那么，请看出人意料的实际结果：

```
[fgao@ubuntu chapter15]# ./a.out
value1 is 0x0
value2 is 0x1
```

为了解释这个意外的结果，我们再次祭出反汇编这一利器：

```
Dump of assembler code for function main:
0x0804841d <+0>:    push    %ebp
0x0804841e <+1>:    mov     %esp,%ebp
0x08048420 <+3>:    and     $0xffffffff,%esp
0x08048423 <+6>:    sub     $0x20,%esp
0x08048426 <+9>:    movl    $0x20,0x14(%esp)
0x0804842e <+17>:   movl    $0x0,0x18(%esp)
0x08048436 <+25>:   mov     0x18(%esp),%eax
0x0804843a <+29>:   mov     %eax,0x4(%esp)
0x0804843e <+33>:   movl    $0x8048510,(%esp)
0x08048445 <+40>:   call    0x80482f0 <printf@plt>
0x0804844a <+45>:   mov     0x14(%esp),%eax
0x0804844e <+49>:   mov     $0x1,%edx
0x08048453 <+54>:   mov     %eax,%ecx
0x08048455 <+56>:   shl     %cl,%edx
0x08048457 <+58>:   mov     %edx,%eax
0x08048459 <+60>:   mov     %eax,0x1c(%esp)
0x0804845d <+64>:   mov     0x1c(%esp),%eax
0x08048461 <+68>:   mov     %eax,0x4(%esp)
0x08048465 <+72>:   movl    $0x8048520,(%esp)
0x0804846c <+79>:   call    0x80482f0 <printf@plt>
0x08048471 <+84>:   mov     $0x0,%eax
0x08048476 <+89>:   leave
0x08048477 <+90>:   ret
End of assembler dump.
```

第6行汇编代码movl\$0x0, 0x18（%esp）对应的C代码为unsigned int value1=1ul<<MOVE_CONSTANT_BITS。也就是说，对于变量value1，编译器直接生成了结果0——该结果也是我们预期的结果。而对于value2，则是真正使用了左移移位指令shl。那么问题就转变成了为什么对一个int整数左移32位，其结果不是0呢？

请看Intel的指令手册中关于shl的说明：

Description

These instructions shift the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to five bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.

现在真相大白了。原来在32位机器上，保存移位个数的指令位只有5位。那么当执行左移32位时，实际上就是左移0位，即没有任何变化。所以value2左移32位时，其值仍然为1。

在32位机器上，实际左移位数等于“指定位数&0x1F”。



说明 在64位机器上，要将Value 1和Value 2修改为long类型左移64位进行测试。